

# Using Machine Learning Tools: Assignment 1

## Overview

In this assignment, you will apply some popular machine learning techniques to the problem of predicting bike rental demand. A data set has been provided containing records of bike rentals in Seoul, collected during 2017-18.

The scenario for this assignment is that you are a new employee of a company (that rents bikes, alongside other activities) and you have been assigned the task of predicting the bike rentals. Your line manager has given you some instructions (those shown below) but is expecting you to be able to do this task without close supervision and to report back with understandable and concise text, graphics and code (and of course the company wants a copy of all the code required to perform this task). Naturally, you are wanting to show that you are a valuable member of the company and although the company allows the use of ChatGPT, you will want to show that you are making useful contributions and that you bring value to the company beyond just being able to type instructions into ChatGPT, as otherwise the company might replace you with a cheaper data entry employee. Hence, you should use ChatGPT whenever you like (or whenever instructed to - see later) but do highlight how your own knowledge and judgement makes a contribution.

The main aims of this assignment are:

- to practice using tools for loading and viewing data sets;
- to check data for common pitfalls and clean it up;
- to plan a simple experiment and prepare the data accordingly;
- to run your experiment and to report and interpret your results clearly and concisely.

This assignment relates to the following ACS CBOK areas: abstraction, design, hardware and software, data and information, HCI and programming.

## General instructions

This assignment is divided into several tasks. Use the spaces provided in this notebook to answer the questions posed in each task. Some questions require writing code, some require graphical results, and some require short comments or analysis as text. It is your

responsibility to make sure your responses are clearly labelled and your code has been fully executed (with the correct results displayed) before submission!

**Do not** manually edit the data set file we have provided! For marking purposes, it's important that your code is written to be able to be run correctly on the original data file.

When creating graphical output, label is clearly, with appropriate titles, xlabel and ylabel, as appropriate.

Most of the tasks in this assignment only require writing a few lines of code! One goal of the assignment is explore [sklearn](#), [pandas](#), [matplotlib](#) and other libraries you will find useful throughout the course, so feel free to use the functions they provide. You are expected to search and carefully read the documentation for functions that you use, to ensure you are using them correctly.

Chapter 2 of the reference book is based on a similar workflow to this prac, so you may look there for some further background and ideas. You can also use any other general resources on the internet that are relevant, including ChatGPT, although do not use someone else's code or answers that directly relate to these questions. If you take a large portion of code or text from the internet or ChatGPT then you should reference where this was taken from, but we do not expect any references for small pieces of code, such as from documentation, blogs or tutorials. Taking, and adapting, small portions of code is expected and is common practice when solving real problems.

The following code imports some of the essential libraries that you will need. You should not need to modify it, but you are expected to import other libraries as needed.

```
In [ ]: # Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

import sklearn
assert sklearn.__version__ >= "0.20"

import pandas as pd
assert pd.__version__ >= "1.0"

# Common imports
import numpy as np
import os

# To plot pretty figures
```

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['axes', labelsizes=14)
mpl.rcParams['xtick', labelsizes=12)
mpl.rcParams['ytick', labelsizes=12)
```

## Step 1: Loading and initial processing of the dataset (40%)

Download the data set `SeoulBikeData.csv` from **MyUni** using the link provided on the assignment page.

The data is stored in a CSV (comma separated values) file and contains the following information

- Date: year-month-day
- Rented Bike Count: Count of bikes rented at each hour
- Hour: Hour of the day
- Temperature: Temperature in degrees Celsius
- Humidity: %
- Windspeed: m/s
- Visibility: 10m
- Dew point temperature: degrees Celsius
- Solar radiation: MJ/m2
- Rainfall: mm
- Snowfall: cm
- Seasons: Winter, Spring, Summer, Autumn
- Holiday: Holiday/No holiday
- Functional Day: NoFunc(Non Functional Hours), Fun(Functional hours)

### 1.1 Load and visualise the data

**Load the data set from the csv file into a DataFrame, summarise it in text using one pandas function, and then visualise each feature with one type of plot (this can be different for each feature).**

In [ ]: *### Your code here*

```
# Load data  
df = pd.read_csv('SeoulBikeData.csv')  
df.head()
```

Out [ ]:

	Date	Rented Bike Count	Hour	Temperature (C)	Humidity (%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature (C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Season
0	01/12/2017	254	0	-5.2	37.0	2.2	2000	-17.6	0	0	0	Winter
1	01/12/2017	204	1	-5.5	38.0	0.8	2000	-17.6	0	0	0	Winter
2	01/12/2017	173	2	-6.0	39.0	1.0	2000	-17.7	0	0	0	Winter
3	01/12/2017	107	3	-6.2	40.0	0.9	2000	-17.6	0	0	0	Winter
4	01/12/2017	78	4	-6.0	36.0	2.3	2000	-18.6	0	0	0	Winter



In [ ]: *# Summarise in text using one pandas function*  
df.info()

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8760 entries, 0 to 8759
Data columns (total 14 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   Date                  8760 non-null   object
 1   Rented Bike Count     8760 non-null   int64
 2   Hour                  8760 non-null   int64
 3   Temperature (C)       8760 non-null   float64
 4   Humidity (%)          8760 non-null   float64
 5   Wind speed (m/s)      8760 non-null   float64
 6   Visibility (10m)      8760 non-null   int64
 7   Dew point temperature (C) 8760 non-null   float64
 8   Solar Radiation (MJ/m2) 8760 non-null   object
 9   Rainfall(mm)          8760 non-null   object
10   Snowfall (cm)         8760 non-null   object
11   Seasons               8760 non-null   object
12   Holiday               8760 non-null   object
13   Functioning Day       8760 non-null   object
dtypes: float64(4), int64(3), object(7)
memory usage: 958.3+ KB

```

**Now, to visualize the data, we make a copy of entire dataframe and work on the copied dataframe**

```

In [ ]: # Make a copy of the original data
        df_vis = df.copy()

```

```

In [ ]: # Check the first few entries of the data
        df_vis.head()

```

Out[ ]:

	Date	Rented Bike Count	Hour	Temperature (C)	Humidity (%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature (C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Season
0	01/12/2017	254	0	-5.2	37.0	2.2	2000	-17.6	0	0	0	Winter
1	01/12/2017	204	1	-5.5	38.0	0.8	2000	-17.6	0	0	0	Winter
2	01/12/2017	173	2	-6.0	39.0	1.0	2000	-17.7	0	0	0	Winter
3	01/12/2017	107	3	-6.2	40.0	0.9	2000	-17.6	0	0	0	Winter
4	01/12/2017	78	4	-6.0	36.0	2.3	2000	-18.6	0	0	0	Winter

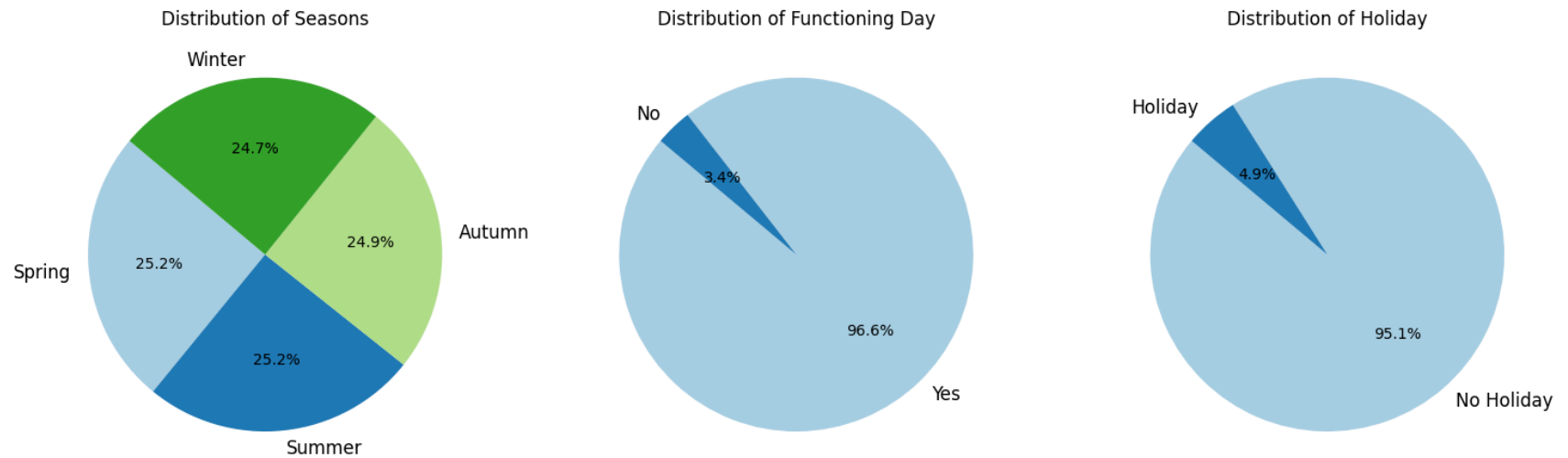
By observing the data, we can see there are **3 categorical features: Seasons, Holiday, Functioning Day**, now we plot these 3 features using **pie chart** to see the category and the percentage of these categories.

```
In [ ]: def plot_pie_charts(df, features, titles, big_title):
    _, axes = plt.subplots(1, 3, figsize=(18, 6))
    for i, feature in enumerate(features):
        counts = df[feature].value_counts()
        axes[i].pie(counts, labels=counts.index, autopct='%1.1f%%', startangle=140, colors=plt.cm.Paired(range(len(counts))))
        axes[i].set_title(titles[i])
    plt.suptitle(big_title, fontsize=20)
    plt.show()

# Features to plot and their titles
features = ['Seasons', 'Functioning Day', 'Holiday']
titles = ['Distribution of Seasons', 'Distribution of Functioning Day', 'Distribution of Holiday']
big_title = 'Visualization of Categorical Features'

# Plot the pie charts
plot_pie_charts(df_vis, features, titles, big_title)
```

## Visualization of Categorical Features



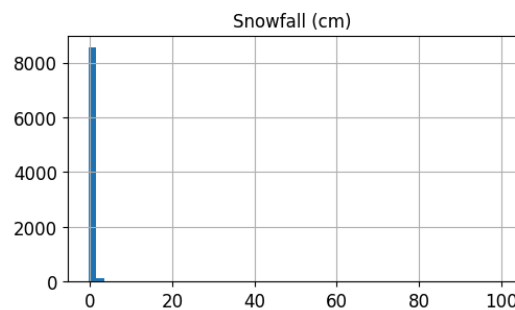
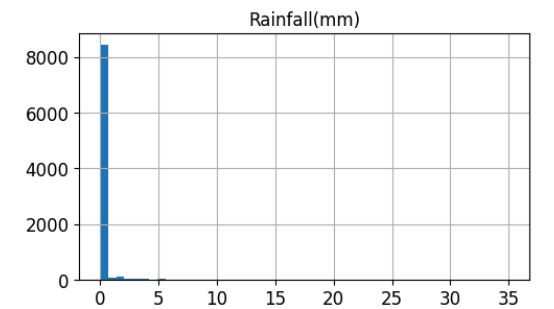
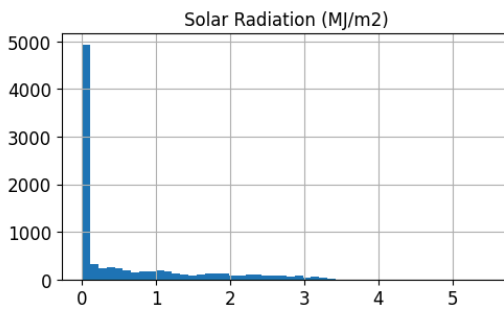
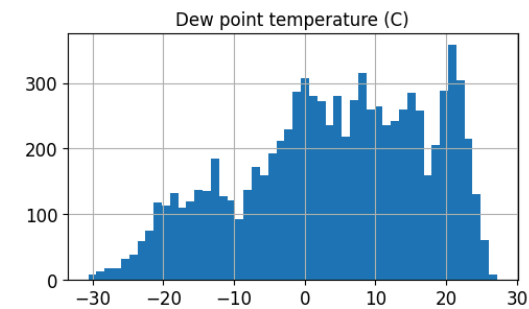
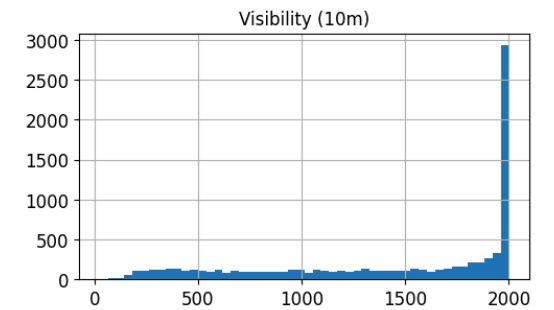
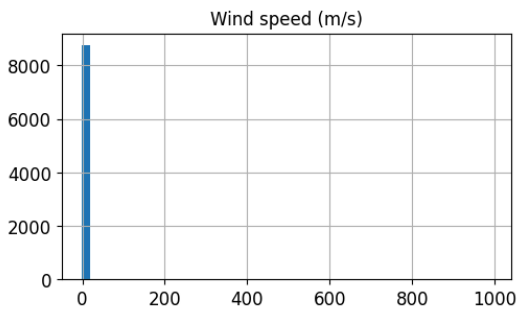
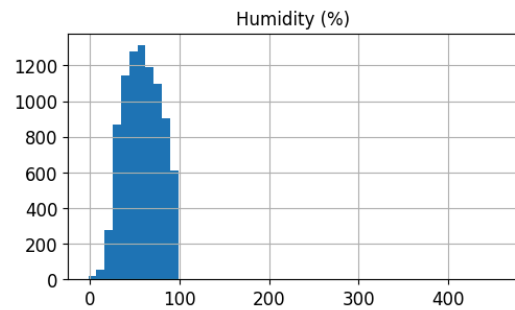
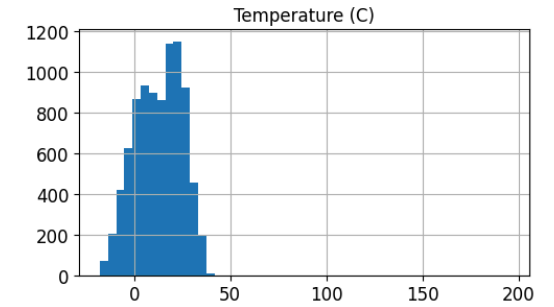
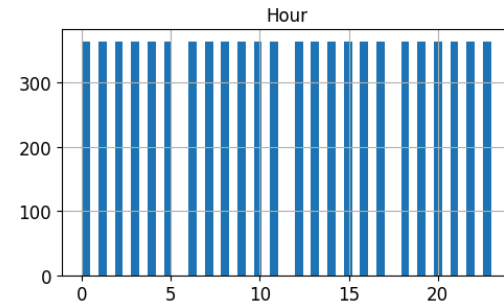
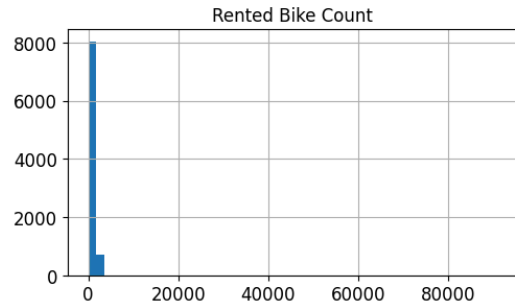
Now, we try to plot the numerical values in histogram, however, from the summary of the data ( `df.info()` ) and the actual data rows ( `df_vis.head()` ), we can see that there are 3 features are actually numerical feature but due to some error inside the data, they are considered as object, which is categorical features. The 3 features are: **Solar Radiation (MJ/m2)**, **Rainfall(mm)**, **Snowfall (cm)**. So now we try to convert the feature value to numeric using `pd.to_numeric()` , then we finally plot all the numerical feature.

```
In [ ]: # Convert to numerical value
numeric_columns = ['Solar Radiation (MJ/m2)', 'Rainfall(mm)', 'Snowfall (cm)']

for column in numeric_columns:
    df_vis[column] = pd.to_numeric(df_vis[column], errors='coerce')
```

```
In [ ]: # Plot numerical features
df_vis.hist(bins=50, figsize=(20, 15))
plt.suptitle('Visualization of Numerical Features', fontsize=20)
plt.show()
```

## Visualization of Numerical Features

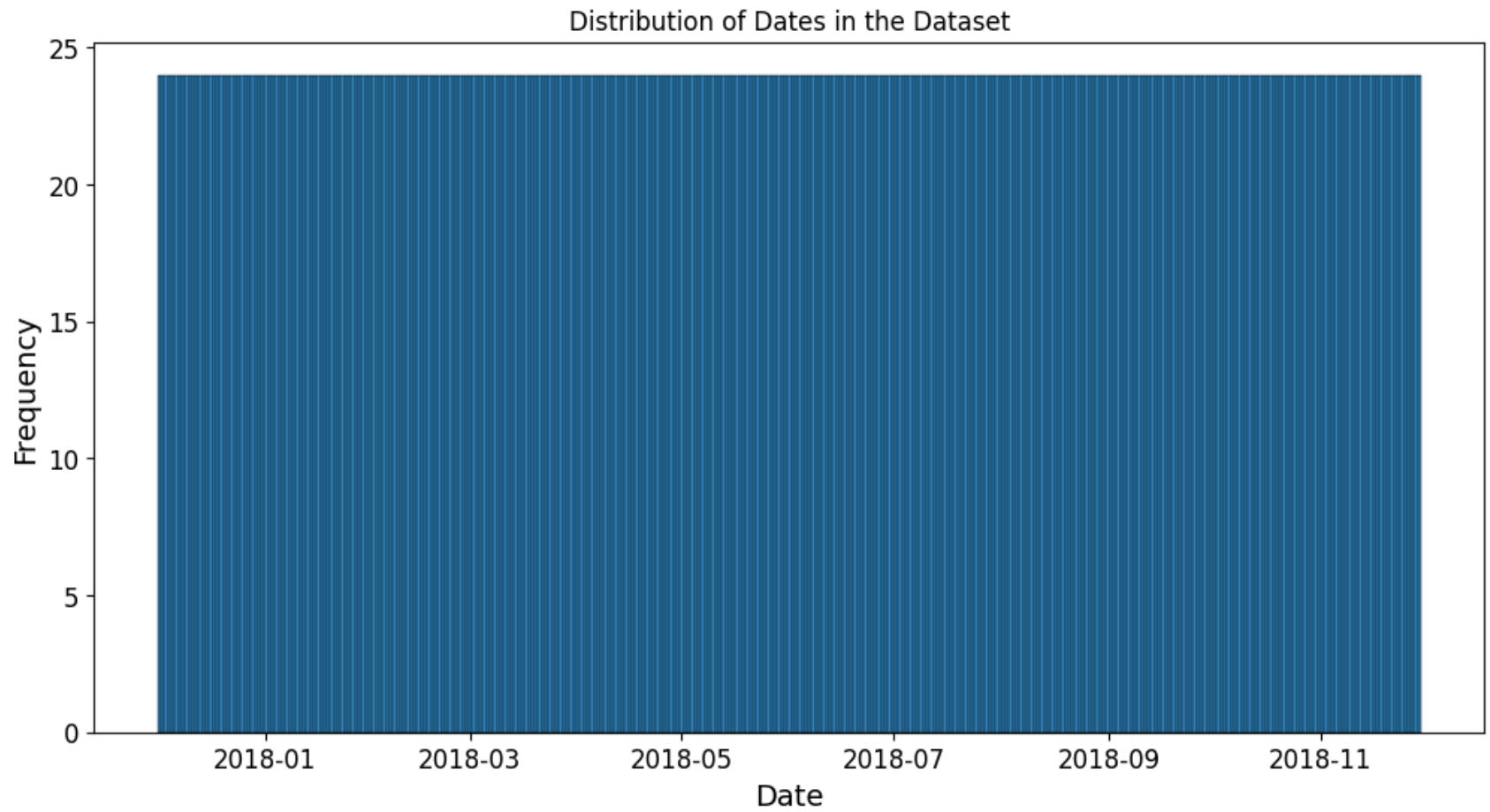




The last feature we need to visualize, Date, is a little bit more weird compare to other features. First we need to convert the Date column to datetime type using `pd.to_datetime()` , and then we plot it in a histogram to see the frequency of each day appear in the dataframe

```
In [ ]: # Convert 'Date' column to datetime
df_vis['Date'] = pd.to_datetime(df_vis['Date'], dayfirst=True)
```

```
In [ ]: # Plot Date as a histogram
plt.figure(figsize=(12, 6))
plt.hist(df_vis['Date'], bins=len(df_vis['Date'].unique()), edgecolor='black', linewidth = 0.3, alpha=0.9)
plt.xlabel('Date')
plt.ylabel('Frequency')
plt.title('Distribution of Dates in the Dataset')
plt.show()
```



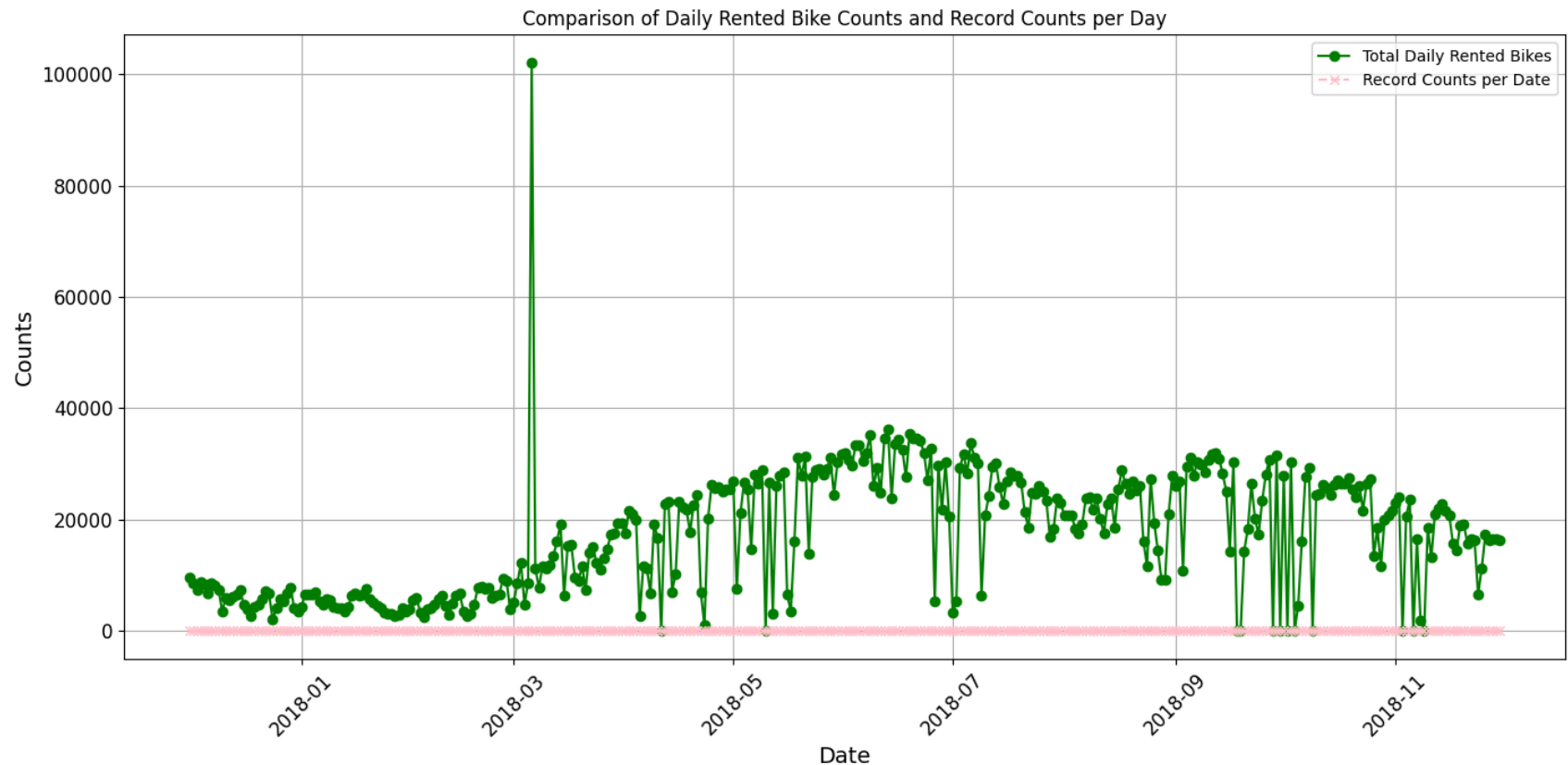
We can see the Date feature contains the day between the end of 2017 till the end of 2018, a whole year range, with equal frequency for each day (24).

And here we show an extra figure to help us have a better understanding of the target feature Rented Bike Count with Date. The following chart shows the number of bike rented per day.

```
In [ ]: date_counts = df_vis['Date'].value_counts().sort_index()

daily_bike_counts = df_vis.groupby('Date')['Rented Bike Count'].sum()
plt.figure(figsize=(14, 7))
plt.plot(daily_bike_counts, marker='o', linestyle='-', color='g', label='Total Daily Rented Bikes')
plt.plot(date_counts, marker='x', linestyle='--', color='pink', label='Record Counts per Date')
```

```
plt.title('Comparison of Daily Rented Bike Counts and Record Counts per Day')
plt.xlabel('Date')
plt.ylabel('Counts')
plt.grid(True)
plt.xticks(rotation=45)
plt.legend()
plt.tight_layout()
plt.show()
```



We can see the number of bike rented per day roughly between 0-40000, however there's on day over 100000, which is an interesting thing to notice, and this is a potential outlier or error we will deal with in later section.

## 1.2 Cleaning the data

Do the following to the data:

- Using the "Functioning day" feature, **remove rows from the DataFrame** where the business is closed and then **delete the Functioning Day feature from the DataFrame**.
- **Convert seasons to a one hot encoded format** (1 binary feature for each of the 4 seasons).
- Replace the **Date** feature with a binary **Weekday** feature (1 for a weekday and 0 for weekend) using the code sample below or your own code.
- **Convert remaining non-numerical features to a numerical format** or replace with NaN (i.e. `np.nan`) where not possible.
- **Identify and fix any outliers and errors in the data.**

Save the result as a new csv file called `CleanedSeoulBikeData.csv` and **upload this** to MyUni along with this notebook when you submit your assignment.

```
In [ ]: ## Example code for weekday feature mapping ##

import datetime
def date_is_weekday(datestring):
    ### return 0 if weekend, 1 if weekday
    dsplit = datestring.split('/')
    wday = datetime.datetime(int(dsplit[2]),int(dsplit[1]),int(dsplit[0])).weekday()
    return int(wday<=4)
```

### 1.2.1 Remove non-functioning day and delete Functioning Day feature

```
In [ ]: ### Your code here (and remember to upload the resulting csv)

# Remove rows where the business is closed and delete the "Functioning Day" feature

df = df[df['Functioning Day'] == 'Yes']
df.drop(columns=['Functioning Day'], inplace=True)
```

### 1.2.2 Convert seasons to a one hot encoded format

```
In [ ]: # Convert Seasons to one-hot encoded format

df = pd.get_dummies(df, columns=['Seasons'])
```

### 1.2.3 Replace the Date feature with a binary Weekday feature

```
In [ ]: # Replace the Weekday feature with a binary weekday feature

df['Weekday'] = df['Date'].apply(date_is_weekday)

df.drop(columns=['Date'], inplace=True)
```

### 1.2.4 Convert remaining non-numerical features to a numerical format

Firstly let's see which columns are still non-numerical features

```
In [ ]: # Show non-numeric columns
non_numeric_columns = df.select_dtypes(exclude=['number']).columns

print(non_numeric_columns)
```

```
Index(['Solar Radiation (MJ/m2)', 'Rainfall(mm)', 'Snowfall (cm)', 'Holiday',
      'Seasons_Autumn', 'Seasons_Spring', 'Seasons_Summer', 'Seasons_Winter'],
      dtype='object')
```

Based on the summary of data( `df.info()` ) and the non-numerical columns, there are 3 steps for us to convert those non-numerical features:

1. convert Holiday feature (since it's actually a binary feature)
2. convert 4 season column (since they are bool type)
3. convert the rest features (since they are actually numerical feature but due to some error in the data it treated as categorical feature)

```
In [ ]: # Check Holiday object value

df['Holiday'].value_counts()
```

```
Out[ ]: Holiday
No Holiday    8057
Holiday       408
Name: count, dtype: int64
```

```
In [ ]: # Convert Holiday feature to numerical format

df['Holiday'] = df['Holiday'].apply(lambda x: 1 if x == 'Holiday' else 0)
```

```
In [ ]: # Convert the one-hot encoded "Seasons" columns from bool to int

season_columns = ['Seasons_Autumn', 'Seasons_Spring', 'Seasons_Summer', 'Seasons_Winter']
for column in season_columns:
    df[column] = df[column].astype(int)
```

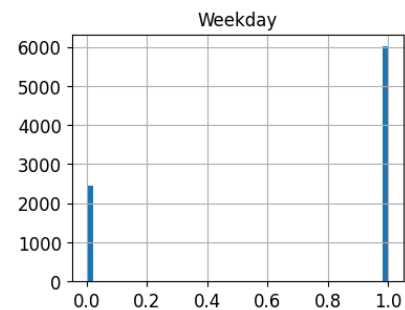
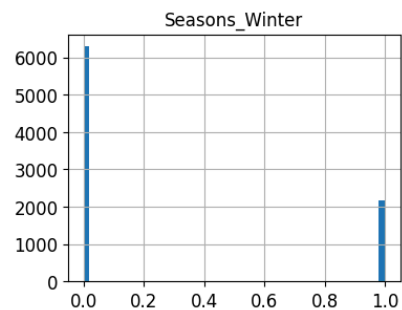
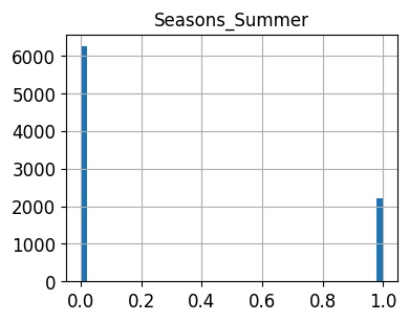
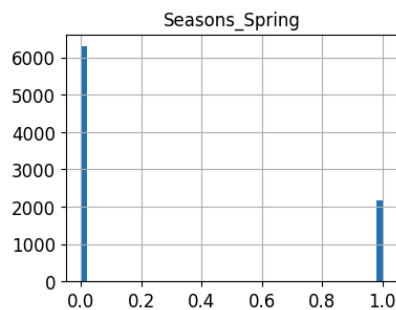
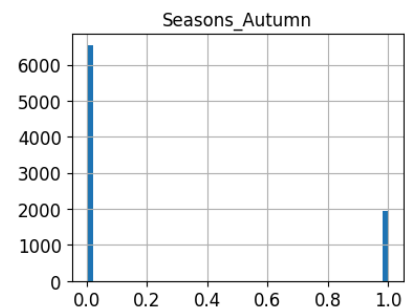
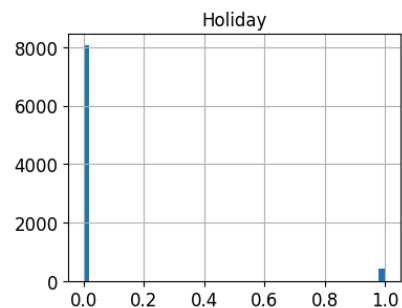
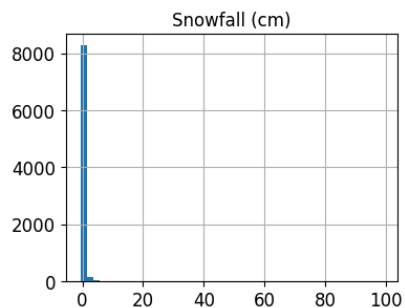
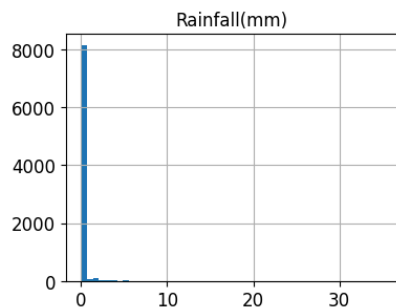
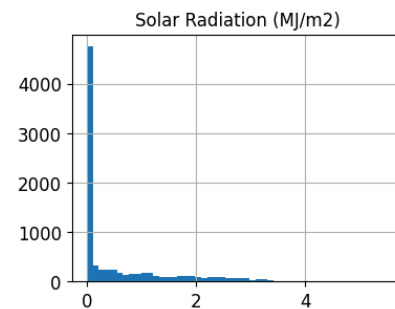
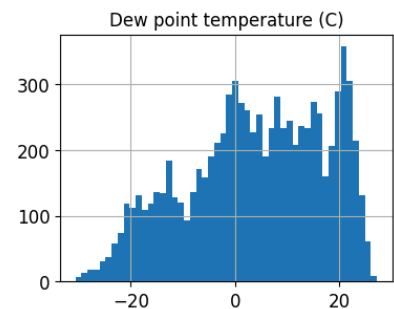
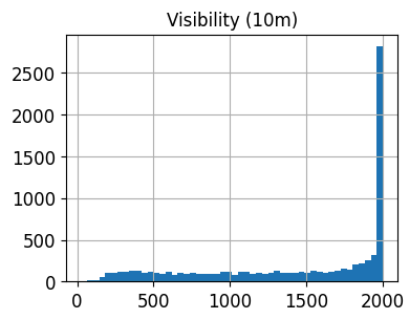
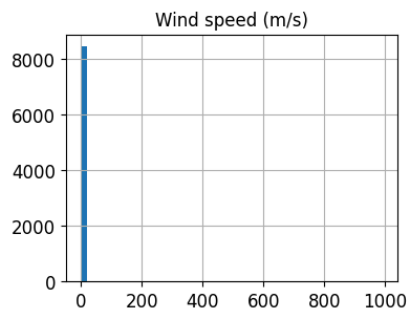
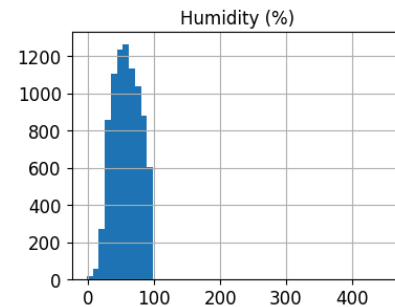
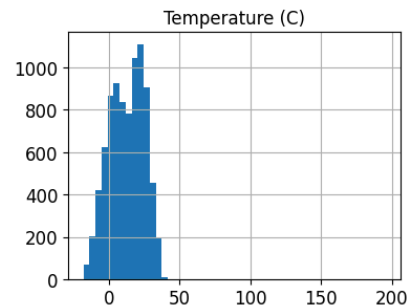
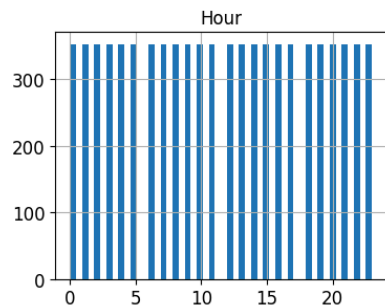
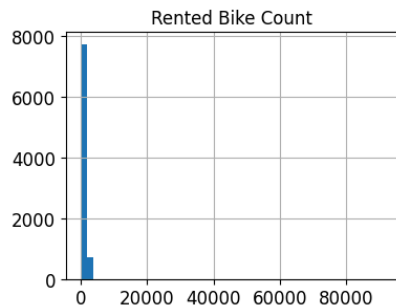
```
In [ ]: # Convert specific features to numeric, replacing non-convertible values with NaN
numeric_columns = ['Solar Radiation (MJ/m2)', 'Rainfall(mm)', 'Snowfall (cm)']

for column in numeric_columns:
    df[column] = pd.to_numeric(df[column], errors='coerce')
```

### 1.2.5 Identify and fix any outliers and errors in the data

```
In [ ]: # Plot all the features
df.hist(bins=50, figsize=(20,15))
plt.suptitle('Plot All Features', fontsize=20)
plt.show()
```

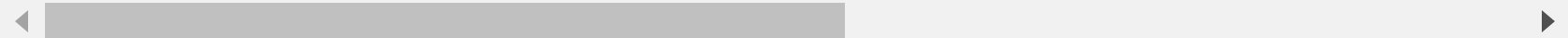
## Plot All Features



```
In [ ]: # Check the statistical description of data
df.describe()
```

```
Out[ ]:
```

	Rented Bike Count	Hour	Temperature (C)	Humidity (%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature (C)	Solar Radiation (MJ/m2)	Rainfall(mm)
count	8465.000000	8465.000000	8465.000000	8465.000000	8465.000000	8465.000000	8465.000000	8464.000000	8444.000000
mean	739.789014	11.507029	12.836090	58.190408	1.854247	1433.873479	3.944997	0.567771	0.149491
std	1172.685973	6.920899	12.545382	20.943092	10.847528	609.051229	13.242399	0.868830	1.126914
min	2.000000	0.000000	-17.800000	-2.200000	-0.700000	27.000000	-30.600000	0.000000	0.000000
25%	214.000000	6.000000	3.000000	42.000000	0.900000	935.000000	-5.100000	0.000000	0.000000
50%	542.000000	12.000000	13.500000	57.000000	1.500000	1690.000000	4.700000	0.010000	0.000000
75%	1084.000000	18.000000	22.700000	74.000000	2.300000	2000.000000	15.200000	0.930000	0.000000
max	90997.000000	23.000000	195.000000	455.000000	991.100000	2000.000000	27.200000	5.520000	35.000000



### Observation of the data features:

From the above histograms and the statistical description of all the features, we found that there are some features contain outlier or error value possibly, as we can see the feature **Rented Bike Count** mainly has value less than 10000, but the max of this feature is 90997. Other features like **Temperature (C)**, **Humidity (%)**, **Wind speed (m/s)**, **Visibility (10m)**, **Solar Radiation (MJ/m2)**, **Rainfall(mm)**, **Snowfall (cm)** since their distribution are seriously skewed.

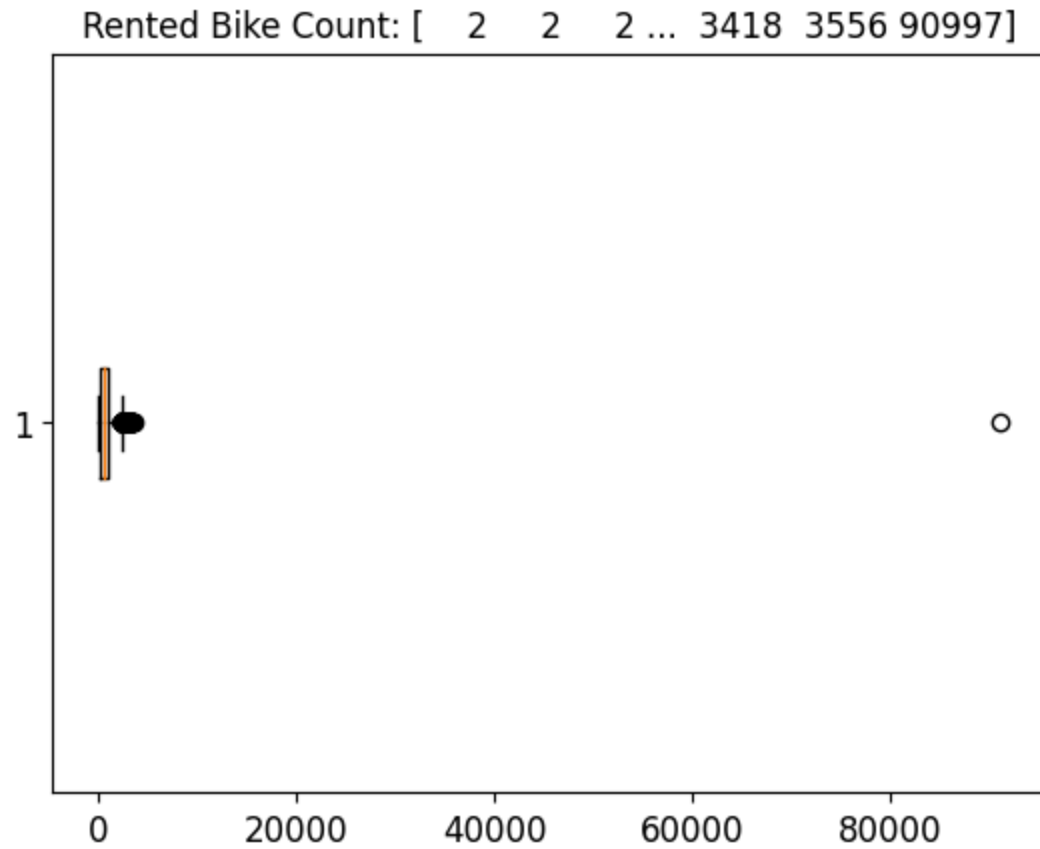
The rest features like Hour, Dew point temperature, Seasons, Holiday, Weekday obviously contains no outlier since they are either normally distributed or they are originally categorical features.

So now we check the box plot and see the sorted data value to detect and remove outlier accurately.

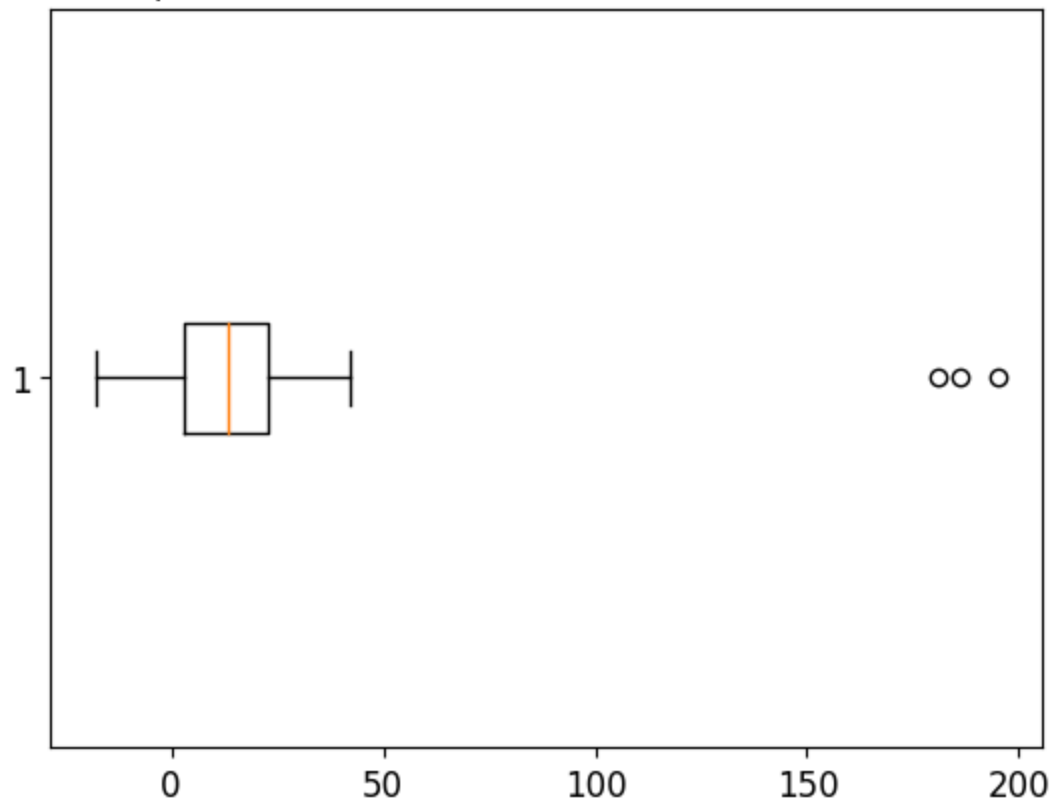
```
In [ ]: possile_error_columns = ['Rented Bike Count', 'Temperature (C)', 'Humidity (%)', 'Wind speed (m/s)', 'Visibility (10m)']
# Plot boxplots excluding NaN values
```



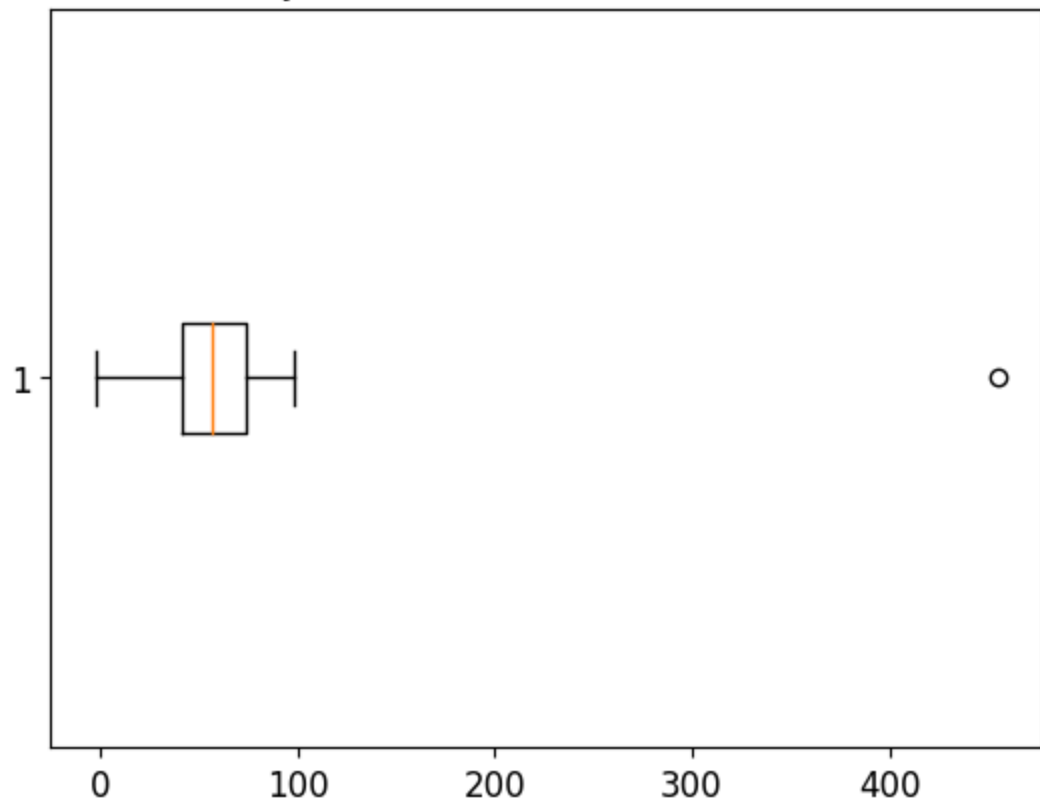
```
for feat in possible_error_columns:
    plt.boxplot(df[feat].dropna(), vert=False)
    plt.title(f'{feat}: {np.sort(df[feat].dropna())}')
    plt.show()
```



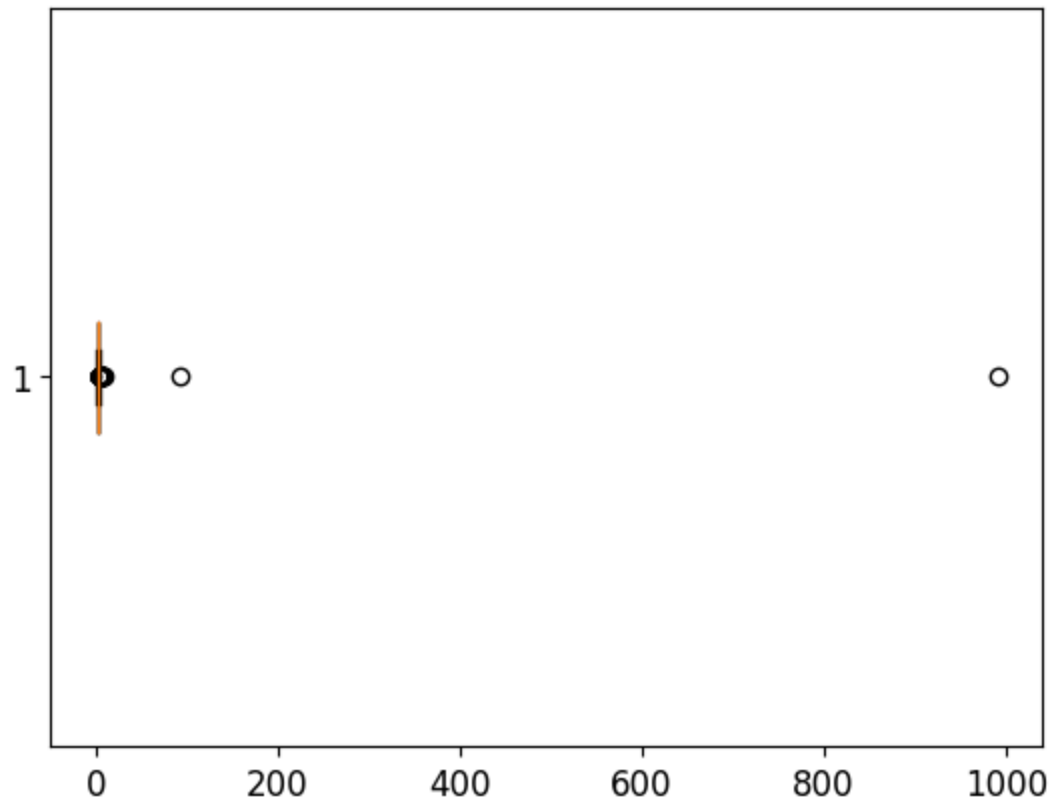
Temperature (C): [-17.8 -17.5 -17.5 ... 181. 186. 195. ]



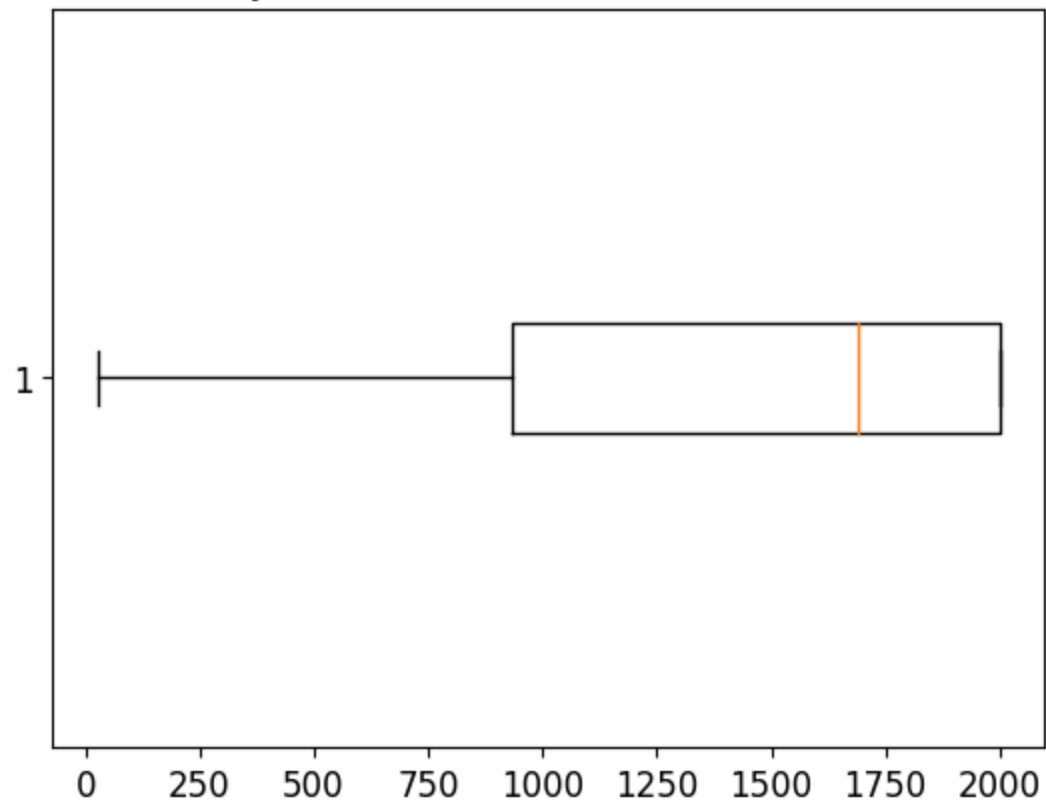
Humidity (%): [ -2.2 0. 0. ... 98. 98. 455. ]



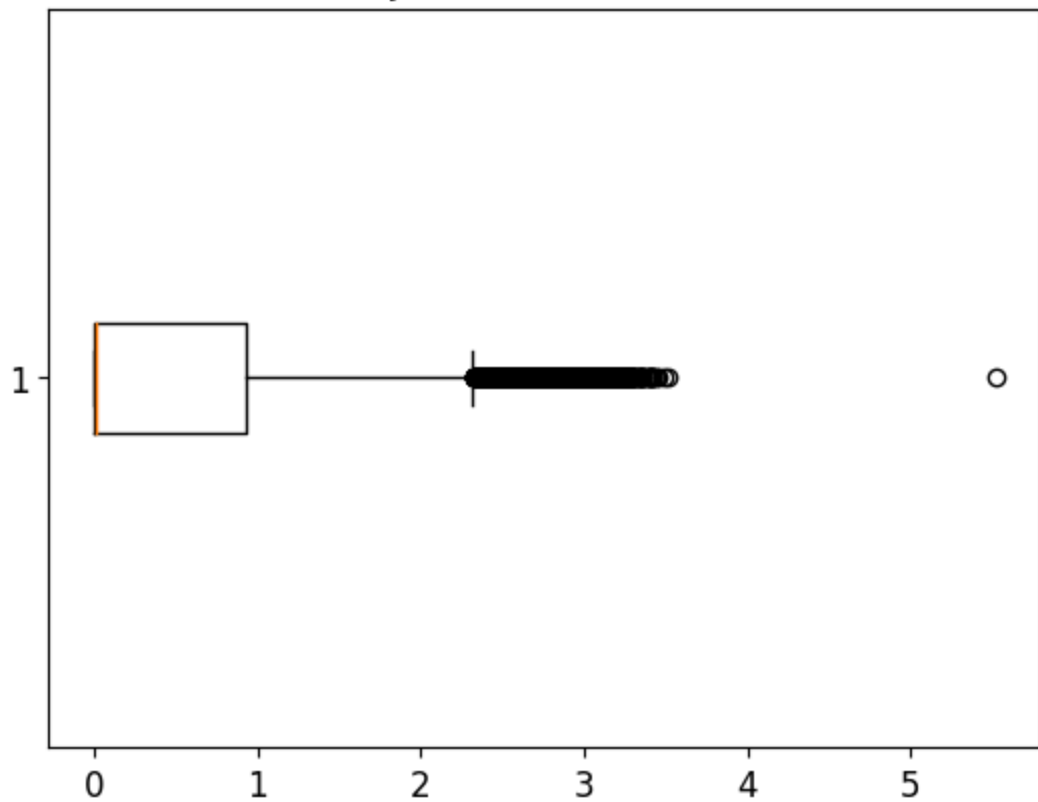
Wind speed (m/s): [-7.000e-01 0.000e+00 0.000e+00 ... 9.200e+00 9.180e+01 9.911e+02]



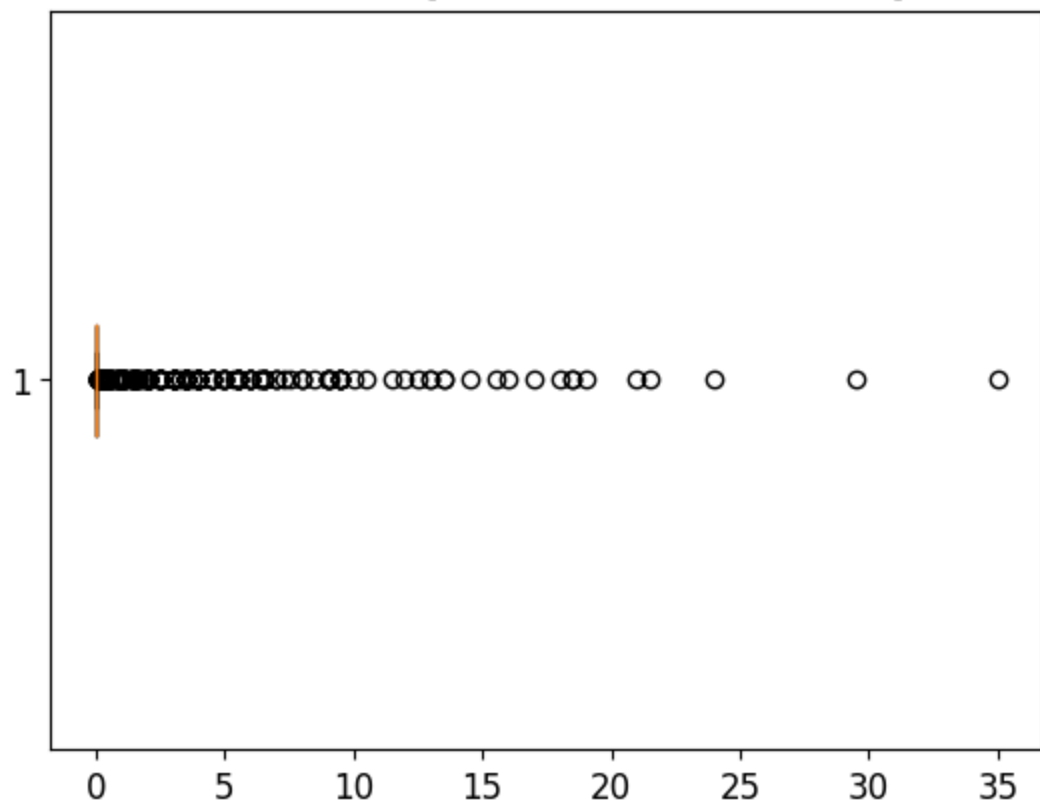
Visibility (10m): [ 27 33 34 ... 2000 2000 2000]

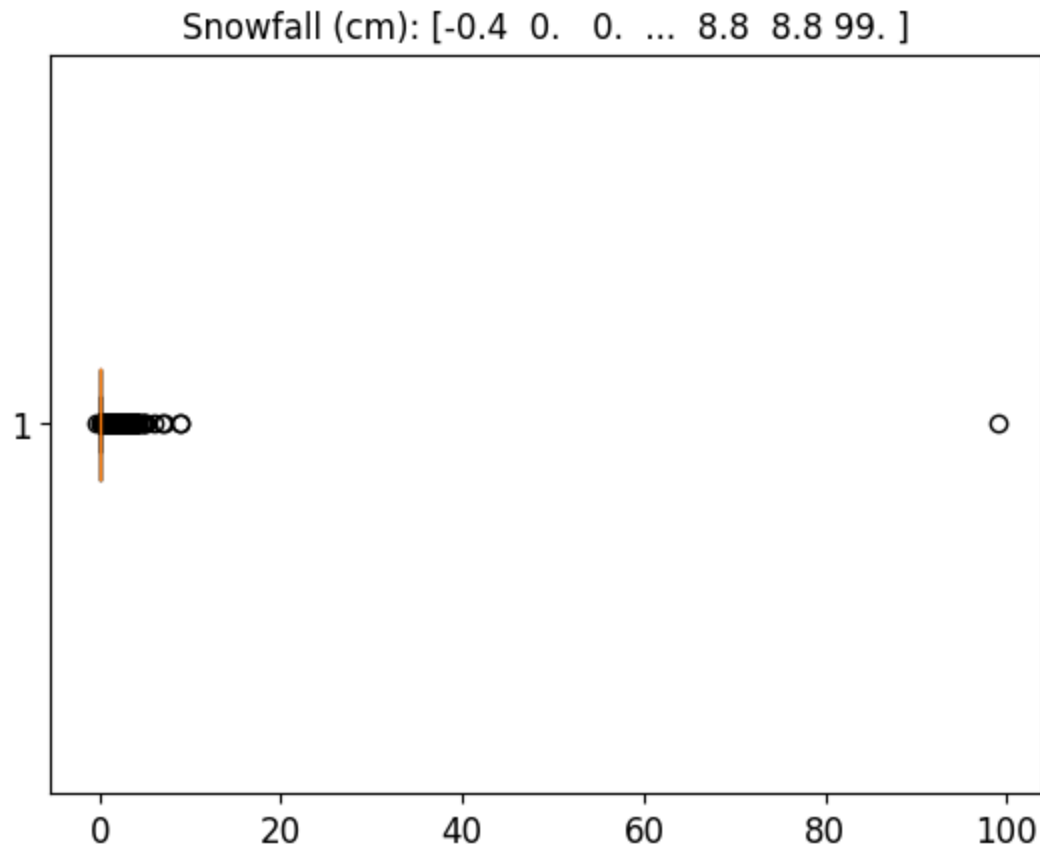


Solar Radiation (MJ/m2): [0. 0. 0. ... 3.49 3.52 5.52]



Rainfall(mm): [ 0. 0. 0. ... 24. 29.5 35. ]





Observation of boxplot (identify the outliers):

- **Rented Bike Count:** we see the largest value of this feature is 90997 while the second largest value is 3556, obviously 90997 is an outlier, then we can set the value which greater than 10000 to nan of this feature.
- **Temperature (C):** we see all the value are less than 100 except 3 largeest value (181, 186, 195), so they are outliers, then we can set the value which greater than 100 to nan of this feature.
- **Humidity (%):** we see there's only one value greater than 100, and since humidity is in percentage, obviously larger than 100 is an error, then we can set the value which greater than 100 to nan of this feature.
- **Wind speed (m/s):** we can see the wind speed are all smaller than 10 except two largest value, which are much larger than the rest value, so they are, thenwe can set the value which greater than 10 to nan of this feature.
- **Visibility (10m):** we can see this feature does not contain apparent outliers



- **Solar Radiation (MJ/m2):** we can see most of the data is 0, and others are all less than 4 except the largest one(5.52), so it's an outlier in this data, now we can set the value which greater than 4 to nan of this feature.
- **Rainfall (mm):** we can see this feature does not contain apparent outliers
- **Snowfall (cm):** we see there's only one value greater than 10 (99), and it's much larger than the rest, also it will be ridiculous if the snowfall in a day is 99 cm, obviously that is an outlier, then we can set the value which greater than 10 to nan of this feature.

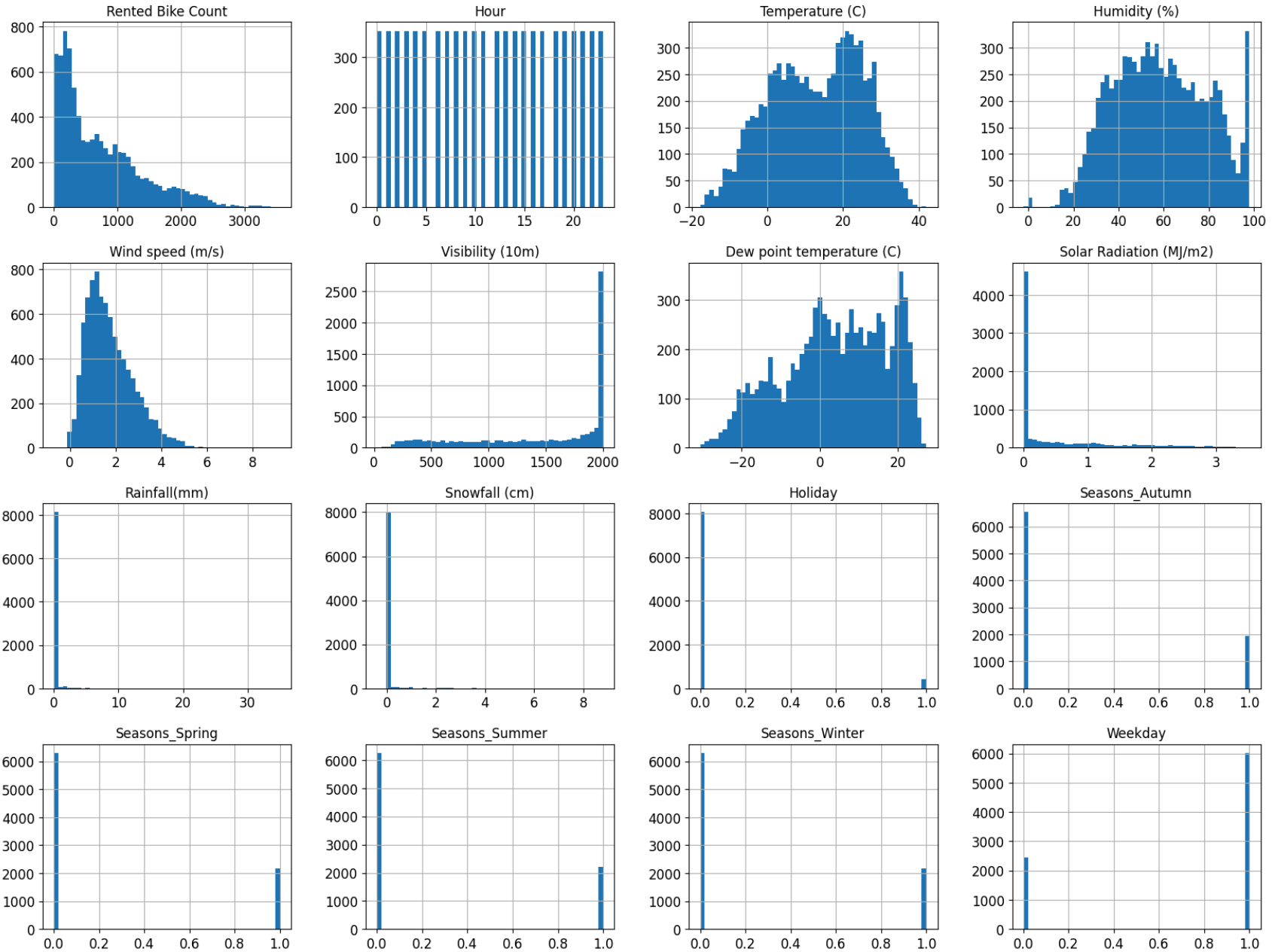
```
In [ ]: # Fixing the identified outliers by replacing them to nan based on the above analysis
```

```
df.loc[df['Rented Bike Count'] > 10000, 'Rented Bike Count'] = np.nan
df.loc[df['Temperature (C)'] > 100, 'Temperature (C)'] = np.nan
df.loc[df['Humidity (%)'] > 100, 'Humidity (%)'] = np.nan
df.loc[df['Wind speed (m/s)'] > 10, 'Wind speed (m/s)'] = np.nan
df.loc[df['Solar Radiation (MJ/m2)'] > 4, 'Solar Radiation (MJ/m2)'] = np.nan
df.loc[df['Snowfall (cm)'] > 10, 'Snowfall (cm)'] = np.nan
```

After dealing with outlier let's check the histogram and the statistical description again of all features.

```
In [ ]: # Plot all the features again
df.hist(bins=50, figsize=(20,15))
plt.suptitle('Plot All Features After fixing outliers', fontsize=20)
plt.show()
```

## Plot All Features After fixing outliers



```
In [ ]: # Check the statistical description of data again
df.describe()
```

Out[ ]:

	Rented Bike Count	Hour	Temperature (C)	Humidity (%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature (C)	Solar Radiation (MJ/m2)	Rainfall(mm)
<b>count</b>	8464.000000	8465.000000	8462.000000	8464.000000	8463.000000	8465.000000	8465.000000	8463.000000	8444.000000
<b>mean</b>	729.125354	11.507029	12.774226	58.143526	1.726728	1433.873479	3.944997	0.567185	0.149491
<b>std</b>	642.382517	6.920899	12.109102	20.495299	1.037826	609.051229	13.242399	0.867212	1.126914
<b>min</b>	2.000000	0.000000	-17.800000	-2.200000	-0.700000	27.000000	-30.600000	0.000000	0.000000
<b>25%</b>	214.000000	6.000000	3.000000	42.000000	0.900000	935.000000	-5.100000	0.000000	0.000000
<b>50%</b>	542.000000	12.000000	13.500000	57.000000	1.500000	1690.000000	4.700000	0.010000	0.000000
<b>75%</b>	1084.000000	18.000000	22.700000	74.000000	2.300000	2000.000000	15.200000	0.930000	0.000000
<b>max</b>	3556.000000	23.000000	42.000000	98.000000	9.200000	2000.000000	27.200000	3.520000	35.000000



As we can see now the data is properly distributed without outliers and the data is finally cleaned.

Now we can save the cleaned data into new csv file.

```
In [ ]: # Check the final summary of the cleaned data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8465 entries, 0 to 8759
Data columns (total 16 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   Rented Bike Count                     8464 non-null   float64
1   Hour                                 8465 non-null   int64
2   Temperature (C)                      8462 non-null   float64
3   Humidity (%)                         8464 non-null   float64
4   Wind speed (m/s)                    8463 non-null   float64
5   Visibility (10m)                    8465 non-null   int64
6   Dew point temperature (C)           8465 non-null   float64
7   Solar Radiation (MJ/m2)             8463 non-null   float64
8   Rainfall(mm)                        8444 non-null   float64
9   Snowfall (cm)                      8443 non-null   float64
10  Holiday                             8465 non-null   int64
11  Seasons_Autumn                      8465 non-null   int32
12  Seasons_Spring                     8465 non-null   int32
13  Seasons_Summer                     8465 non-null   int32
14  Seasons_Winter                     8465 non-null   int32
15  Weekday                             8465 non-null   int64
dtypes: float64(8), int32(4), int64(4)
memory usage: 992.0 KB
```

```
In [ ]: # Save the cleaned data to a new CSV file
        cleaned_file_path = 'CleanedSeoulBikeData.csv'
        df.to_csv(cleaned_file_path, index=False)
```

## Step 2: Pre-process the data and perform the first fit (20%)


### 2.1 Imputation and Pre-Processing

Make sure that you have set any problematic values in the numerical data to `np.nan` and then write code for a **sklearn pipeline that will perform imputation** to replace problematic entries (nan values) with an appropriate **median** value \*\*\*and\* do any other pre-processing\*\* that you think should be used.

```
In [ ]: # Check statistic of features again
        df.describe()
```

Out[ ]:

	Rented Bike Count	Hour	Temperature (C)	Humidity (%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature (C)	Solar Radiation (MJ/m2)	Rainfall(mm)
count	8464.000000	8465.000000	8462.000000	8464.000000	8463.000000	8465.000000	8465.000000	8463.000000	8444.000000
mean	729.125354	11.507029	12.774226	58.143526	1.726728	1433.873479	3.944997	0.567185	0.149491
std	642.382517	6.920899	12.109102	20.495299	1.037826	609.051229	13.242399	0.867212	1.126914
min	2.000000	0.000000	-17.800000	-2.200000	-0.700000	27.000000	-30.600000	0.000000	0.000000
25%	214.000000	6.000000	3.000000	42.000000	0.900000	935.000000	-5.100000	0.000000	0.000000
50%	542.000000	12.000000	13.500000	57.000000	1.500000	1690.000000	4.700000	0.010000	0.000000
75%	1084.000000	18.000000	22.700000	74.000000	2.300000	2000.000000	15.200000	0.930000	0.000000
max	3556.000000	23.000000	42.000000	98.000000	9.200000	2000.000000	27.200000	3.520000	35.000000



Here, except the basic imputation, I also added standardalization into my pipeline since all the features are not in the same scale as we can see by `df.describe()` .

Standardizing data is beneficial when performing regression analysis for several reasons:

- **Improving Model Convergence:** Many regression algorithms, particularly those involving iterative optimization processes like gradient descent, converge faster when the features are standardized. Standardizing the features ensures that they are on a similar scale, preventing any one feature from dominating the optimization process.
- **Handling Different Scales:** In datasets where features have different units and scales (e.g., age in years vs. income in dollars), features with larger scales can disproportionately influence the model's predictions. Standardization mitigates this by scaling all features to have a mean of zero and a standard deviation of one, ensuring that each feature contributes equally to the model.
- **Interpretability of Coefficients:** Standardized data allows for easier interpretation of the regression coefficients. In standardized regression, the coefficients represent the change in the response variable for a one standard deviation change in the predictor variable, making it easier to compare the effects of different predictors.

In conclusion, Standardizing data is a crucial preprocessing step in regression analysis, especially when dealing with features of varying scales. It enhances model performance, interpretability, and convergence, making the regression results more reliable and

meaningful.

```
In [ ]: from sklearn.pipeline import Pipeline
        from sklearn.impute import SimpleImputer
        from sklearn.preprocessing import StandardScaler

        # Create a pipeline for imputation and scaling
        preprocess_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler()),
        ])
```

The above code perform inputation seperately for df['Rented Bike Count'] column, and now we can perfrom first fit.

## 2.2 Predicting bike rentals

A regression approach will be used for this problem: that is, "bike rentals" will be treated as a real number whose value will be predicted. If necessary, it could be rounded to the nearest integer afterwards, but this will not be necessary here. The root mean squared error (RMSE) metric will be used to quantify performance.

**Split the data** appropriately so that 20% of it will be kept as a hold-out test set. **Using the pipeline** you wrote above, pre-process and fit a **\*\*\*linear regression\* model\*\*** to the data in an appropriate way. After this, **calculate and print the RMSE of the fit to the training data**.

To act as a simple baseline for comparison purposes, **also calculate and print the RMSE** that you would get if *all* the predictions were set to be the **mean of the training targets** (i.e. bike rentals).

```
In [ ]: ### Your code and outputs here

        from sklearn.model_selection import train_test_split
        from sklearn.linear_model import LinearRegression
        from sklearn.metrics import mean_squared_error

        # Separate features and target
        X = df.drop(columns=['Rented Bike Count'])
        y = df['Rented Bike Count']
```

```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
```

Here there's an extra problem, the target value `Rented Bike Count` has NaN value that we should impute, however if we simply do the following

```
preprocess_pipeline.fit(X_train, y_train)
```

The preprocess pipeline does not handle missing values in `y_train`. The `SimpleImputer` and `StandardScaler` steps only apply to the features `X_train`, not the target `y_train`. If `y_train` contains missing values, we need to handle them separately before fitting the pipeline. The same happened on the `y_test`, we should perform imputation on `y_test` too, the reason why we don't impute missing values before splitting data, is because we will have risk leaking information from test set into training set. This can artificially inflate the performance metrics of our model because it has seen information from the test set during training.

```
In [ ]: # Check the NaN value in both y_train and y_test
print(f"There have {y_train.isna().sum()} NaN in y_train")
print(f"There have {y_test.isna().sum()} NaN in y_test")
```

There have 0 NaN in `y_train`

There have 1 NaN in `y_test`

```
In [ ]: # Create the imputer with the median strategy
y_imputer = SimpleImputer(strategy='median')

# Fit and transform on y_train
y_train = y_imputer.fit_transform(y_train.to_numpy().reshape(-1, 1))

# Transform y_test (avoid data leakage)
y_test = y_imputer.transform(y_test.to_numpy().reshape(-1, 1))

# Convert y back to pandas Series with the correct index
y_train = pd.Series(y_train.flatten(), index=X_train.index, name='Rented Bike Count')
y_test = pd.Series(y_test.flatten(), index=X_test.index, name='Rented Bike Count')
```

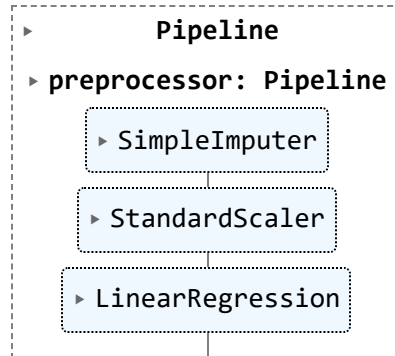
After this we can now create the final pipeline and feed the data into this final pipeline to perform first fit

```
In [ ]: # Make the entire pipeline
pipeline = Pipeline([
    ('preprocessor', preprocess_pipeline),
    ('regressor', LinearRegression())
```

```
])
```

```
# Fit the pipeline to the training data  
pipeline.fit(X_train, y_train)
```

Out[ ]:



```
In [ ]: # Predict on the training data  
y_train_pred = pipeline.predict(X_train)  
  
# Calculate RMSE for the training data  
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))  
print(f'Training RMSE: {rmse_train:.2f}')  
  
# Baseline model: Mean of training targets  
baseline_pred = np.full_like(y_train, y_train.mean())  
baseline_rmse = np.sqrt(mean_squared_error(y_train, baseline_pred))  
print(f'Baseline RMSE: {baseline_rmse:.2f}')
```

Training RMSE: 437.26

Baseline RMSE: 646.17

We also have a look at the test RMSE

```
In [ ]: # Predict on the testing data  
y_test_pred = pipeline.predict(X_test)  
  
# Calculate RMSE for the test set  
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))  
print(f'Testing RMSE: {test_rmse:.2f}')
```

Testing RMSE: 416.67



After the first fit, we can see the RMSE on the training data is around 437, which is a not bad result, compared to the baseline RMSE. Now we need to consider several things to improve the performance:

- Consider other more powerful regression model
- Do hyper-parameter optimisation
- Try cross-validation

## Step 3: Hyper-parameter optimisation (30%)

**Use ChatGPT** (along with any modifications that you require) to create and run code (using sklearn pipelines) that will do the following:

- fit a **linear regression** and a **Support Vector Regression** method to the data using **10-fold cross validation** for each model
- display the **mean and standard deviation** of the **RMSE values** for each model (at baseline) in the *appropriate datasets*
- perform a **hyper-parameter optimisation** on each model using **GridSearch**
- display the **mean and standard deviation** of the **RMSE values** for each model (after optimisation) in the *appropriate datasets*
- choose the **best model** and **visualise the results** with a single graphic of your choice

**Display the ChatGPT prompt** and the **code**, *including any fixes* that you needed to make to get the code to work, along with the **outputs** obtained by running the code.

Some decision and explanation before using ChatGPT:

- Notice we need to display the RMSE in the appropriate dataset before and after optimisation, and visualise the results of best model in the end, I decided to do the cross-validation on the training set only and test on the test data in the end then visualise it, the reasons are:
  - Performing cross-validation and grid search on the training set prevents the model from overfitting to the entire dataset, ensuring it learns general patterns rather than memorizing specific data points. If we do cross-validation on the entire set, we will have no proper test set to test the final performance/result.
- Notice in the gridsearch, there may have some warning messages indicate the combination of parameter is not valid (for example using gamma parameter with linear kernel in SVR), to get rid of the warning we filter the warning messages in the beginning.

Also the model I use is ChatGPT 4o

### 3.1 My ChatGPT prompt

Now we have a dataframe called df which contains cleaned data, all the features are numerical, the task is to use other features to train a regression model to predict the target feature called 'Rented Bike Count'.

Now, we have the pipeline already, which contains two part, the first part is the preprocess\_pipeline:

```
preprocess_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
])
```

and the second part should be the combination of first pipeline and a regression model. You don't need to split the data again since I already prepared the data for you, they are `X_train, X_test, y_train, y_test`

Now perform the following task:

- fit a linear regression and a Support Vector Regression method to the data using 10-fold cross validation for each model
- display the mean and standard deviation of the RMSE values for each model (at baseline) in the training datasets
- perform a hyper-parameter optimisation on each model using GridSearch, for linear regression consider fit\_intercept and positive, for Support Vector Regression consider C, kernel, epsilon. In the end tell me the best parameter set of each model
- display the mean and standard deviation of the RMSE values for each model (after optimisation) in the training datasets
- choose the best model and visualise the results with a single graphic of your choice

Filter all the warning out do not display them.

### 3.2 ChatGPT code with outputs (after fixing)

```
In [ ]: ### Code here (with outputs)

from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
```

```

from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.metrics import make_scorer, mean_squared_error
from sklearn.model_selection import cross_val_score, GridSearchCV
import warnings

# Filter warnings
warnings.filterwarnings('ignore')

# Preprocessing Pipeline
preprocess_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
])

# Set up pipelines for both models
pipe_lr = Pipeline([
    ('preprocessor', preprocess_pipeline),
    ('regressor', LinearRegression())
])

pipe_svr = Pipeline([
    ('preprocessor', preprocess_pipeline),
    ('regressor', SVR())
])

```

```

In [ ]: # Define RMSE as a scorer
def rmse_score(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

rmse_scorer = make_scorer(rmse_score, greater_is_better=False)

# Perform 10-fold cross-validation
cv_scores_lr = cross_val_score(pipe_lr, X_train, y_train, cv=10, scoring=rmse_scorer)
cv_scores_svr = cross_val_score(pipe_svr, X_train, y_train, cv=10, scoring=rmse_scorer)

print("Linear Regression - mean RMSE:", -cv_scores_lr.mean(), "Std Dev:", cv_scores_lr.std())
print("SVR - mean RMSE:", -cv_scores_svr.mean(), "Std Dev:", cv_scores_svr.std())

```

Linear Regression - mean RMSE: 438.532065028831 Std Dev: 15.651069453529836  
SVR - mean RMSE: 540.7671160152353 Std Dev: 20.08963395621336

```
In [ ]: # Hyperparameter tuning
param_grid_lr = {
    'regressor__fit_intercept': [True, False],
    'regressor__positive': [True, False]
}

param_grid_svr = {
    'regressor__C': [0.1, 1, 10],
    'regressor__kernel': ['linear', 'rbf'],
    'regressor__gamma': ['scale', 'auto']
}

# Grid search and show best parameter
grid_lr = GridSearchCV(pipe_lr, param_grid_lr, cv=10, scoring=rmse_scorer)
grid_svr = GridSearchCV(pipe_svr, param_grid_svr, cv=10, scoring=rmse_scorer)

grid_lr.fit(X_train, y_train)
print("Best params LR:", grid_lr.best_params_)

grid_svr.fit(X_train, y_train)
print("Best params SVR:", grid_svr.best_params_)
```

Best params LR: {'regressor\_\_fit\_intercept': True, 'regressor\_\_positive': False}  
 Best params SVR: {'regressor\_\_C': 10, 'regressor\_\_gamma': 'scale', 'regressor\_\_kernel': 'rbf'}

```
In [ ]: # ChatGPT did not show me the code of this cell for some reason, this is added by myself

# Display RMSE for Linear Regression after GridSearch
lr_cv_results = grid_lr.cv_results_
lr_rmse_mean = -lr_cv_results['mean_test_score'][grid_lr.best_index_]
lr_rmse_std = lr_cv_results['std_test_score'][grid_lr.best_index_]
print(f'Optimized Linear Regression RMSE: Mean = {lr_rmse_mean:.2f}, Std Dev = {lr_rmse_std:.2f}, Params = {grid_lr.best_params_}')

# Display RMSE for SVR after GridSearch
svr_cv_results = grid_svr.cv_results_
svr_rmse_mean = -svr_cv_results['mean_test_score'][grid_svr.best_index_]
svr_rmse_std = svr_cv_results['std_test_score'][grid_svr.best_index_]
print(f'Optimized SVR RMSE: Mean = {svr_rmse_mean:.2f}, Std Dev = {svr_rmse_std:.2f}, Params = {grid_svr.best_params_}')
```

Optimized Linear Regression RMSE: Mean = 438.53, Std Dev = 15.65, Params = {'regressor\_\_fit\_intercept': True, 'regressor\_\_positive': False}  
Optimized SVR RMSE: Mean = 397.41, Std Dev = 14.48, Params = {'regressor\_\_C': 10, 'regressor\_\_gamma': 'scale', 'regressor\_\_kernel': 'rbf'}

```
In [ ]: # Choose the best model
if -grid_lr.best_score_ < -grid_svr.best_score_:
    best_model = grid_lr.best_estimator_ # ChatGPT gives me
else:
    best_model = grid_svr.best_estimator_

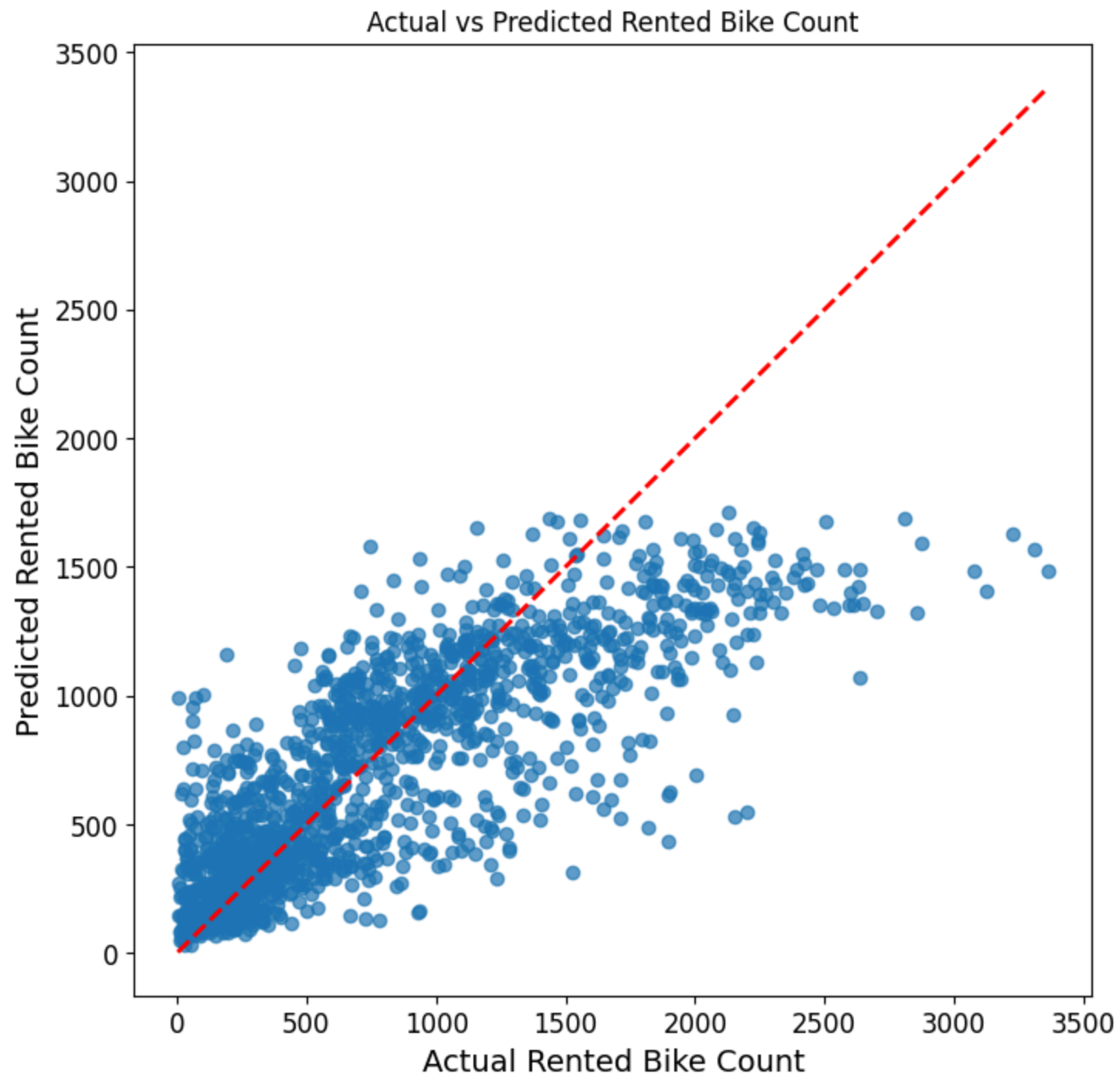
# Fit the best model on the entire training set
best_model.fit(X_train, y_train)

# Predict on the test set
y_test_pred = best_model.predict(X_test)

# Calculate RMSE for the test set
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
print(f'Test RMSE of the Best Model: {test_rmse:.2f}')
```

Test RMSE of the Best Model: 367.97

```
In [ ]: # Visualize the results
plt.figure(figsize=(8, 8))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel('Actual Rented Bike Count')
plt.ylabel('Predicted Rented Bike Count')
plt.title('Actual vs Predicted Rented Bike Count')
plt.show()
```



### Observation:

- We can see after performing 10-cross validation, the mean RMSE of SVR (397.41) is lower than Linear Regression (438.53), suggests that SVR perform better than Linear Regression. So we choose SVR as our best model, and the best parameter is C: 10, gamma: scale, kernel: rbf
- The test RMSE of SVR is even better than mean cross-validation RMSE, means the model generalize well, the model do not have obvious overfitting issue.
- By observing the comparison chart of the predicted value vs actual value of rented bike count, we can see value-pair scatter plot, gather near  $y=x$  line, which means the predicted value is near actual value. However, as we can see the model does not perform very well when the actual rented bike count is larger ( $>2000$ ), the model tends to predict a lower value than actual value. That's means the model may need further tuning, or need more data, or maybe we need to use more powerful models like nerual network etc.

## 3.3 Details of my fixing for the ChatGPT code to make it runnable

There are only one place I need to fix/add to make sure the code runnable and complete all the task: In the beginning ChatGPT did not give me the code of displaying the RMSE mean/std dev after optimisation although in our prompt I let it help me doing this.

The code I add to show RMSE is:

```
# Display RMSE for Linear Regression after GridSearch
lr_cv_results = grid_lr.cv_results_
lr_rmse_mean = -lr_cv_results['mean_test_score'][grid_lr.best_index_]
lr_rmse_std = lr_cv_results['std_test_score'][grid_lr.best_index_]
print(f'Optimized Linear Regression RMSE: Mean = {lr_rmse_mean:.2f}, Std Dev = {lr_rmse_std:.2f}, Params
= {grid_lr.best_params_}')
```

```
# Display RMSE for SVR after GridSearch
svr_cv_results = grid_svr.cv_results_
svr_rmse_mean = -svr_cv_results['mean_test_score'][grid_svr.best_index_]
svr_rmse_std = svr_cv_results['std_test_score'][grid_svr.best_index_]
print(f'Optimized SVR RMSE: Mean = {svr_rmse_mean:.2f}, Std Dev = {svr_rmse_std:.2f}, Params =
{grid_svr.best_params_}')
```

This code grab the mean RMSE and the standard deviation of RMSE of the best model we found in the grid search

## Step 4: Further improvements (10%)

Consider the code that you obtained from ChatGPT above and find one error, or one thing that could be improved, or one reasonable alternative (even if it might not necessarily lead to an improvement). **Describe this error/improvement/alternative in the box below.**

### Improvement/Alternative: Simplify the Calculation of RMSE Using `neg_root_mean_squared_error`

In the previous code, we converted negative mean squared error (MSE) scores to root mean squared error (RMSE) scores manually by making a scorer. However, scikit-learn provides a built-in scorer, `neg_root_mean_squared_error`, which directly computes the negative RMSE, simplifying the process further.

Here's how we can use `neg_root_mean_squared_error` to perform cross-validation and directly obtain RMSE scores:

```
# Perform 10-fold cross-validation using neg_root_mean_squared_error
cv_scores_lr = cross_val_score(pipe_lr, X_train, y_train, cv=10, scoring='neg_root_mean_squared_error')
cv_scores_svr = cross_val_score(pipe_svr, X_train, y_train, cv=10, scoring='neg_root_mean_squared_error')

# Convert negative RMSE scores to positive RMSE scores
rmse_scores_lr = -cv_scores_lr
rmse_scores_svr = -cv_scores_svr

print(f"RMSE scores for Linear Regression: {rmse_scores_lr}")
print(f"RMSE scores for SVR: {rmse_scores_svr}")
```

#### Explanation:

- **Direct RMSE Calculation:** Using `neg_root_mean_squared_error` directly computes the RMSE scores during cross-validation, eliminating the need for manual conversion from MSE to RMSE.
- **Simplified Code:** This approach reduces complexity and potential errors, making the code more readable and concise. By leveraging `neg_root_mean_squared_error`, we can streamline your cross-validation process and focus more on analyzing the results rather than handling intermediate calculations.

**Conclusion:** Using `neg_root_mean_squared_error` as scorer directly actually does not lead to any significant improvement