bike_sharing_systems_final_project_10_18

October 18, 2024

1 Bike Sharing Sytems:

converting casual users to registered users as a key factor in determining bike demand.

1.1 Contributors:

Swathi Subramanyam Pabbathi

Lakshmi Deepthi Pamula

Tej Singh

2 Introduction:

"In the United States, public bicycle share programs have largely centered around major cities and universities.Some corporate campuses have private systems.According to a report by the National Association of City Transportation Officials, a total of 35 million bike-share trips took place within the United States in 2017 across 100 bike-share systems across the country, operated by eight companies."

Bike-sharing systems, like other public transportation options such as buses, trains, and taxis, experience fluctuating user demand based on various factors. Accurately predicting this demand is crucial for ensuring the availability of bikes at docking stations, making the service more reliable and convenient for users. However, beyond simply forecasting demand, a key objective is to understand how to convert casual riders into registered users, which is essential for long-term business growth.

3 Problem Statement:

This project focuses on leveraging predictive models to estimate bike demand and identify the factors that influence a user's decision to transition from casual use to becoming a registered member. By doing so, bike sharing operators can proactively manage fleet distribution while also implementing targeted strategies to drive customer loyalty and increase recurring revenue.

The findings from this analysis will demonstrate the significance of understanding both demand trends and user conversion, offering a business case for how bike sharing systems can improve customer retention, optimize revenue, and enhance their overall service offering.

Leveraging the Bike Sharing Demand dataset from Kaggle, we aim to predict two key bike-sharing metrics casual and registered bike rentals and derive a new metric called the conversion ratio, which measures how effectively casual riders are converting into registered users.

Predicting these demands can prove to be efficacious as it allows one to stock bikes in docking stations according to user demands in advance.It allows bike sharing systems to become not just an economical and healthy mode of transport, but also a reliable mode of transport.

4 Business Benefits

Optimized Bike Distribution:

Predicting bike demand, especially for casual users, ensures that the right number of bikes is available at different locations, minimizing stockouts or over-supply issues.

Improved Marketing Efficiency:

By leveraging the conversion ratio, marketing efforts can be directed toward casual users during key time periods (e.g., holidays, weekends) to convert them into registered users, resulting in more sustained business growth.

Enhanced Customer Retention:

Understanding what drives casual users to become registered users enables the business to refine its retention strategies. A focus on increasing the conversion ratio directly supports customer loyalty and long-term engagement.

Revenue Growth and Stability:

The model's ability to predict the conversion ratio and bike rentals helps the company boost its revenue by converting casual users into more frequent registered users, ensuring a stable and consistent income stream.

[1]:
```
import pandas as pd
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

5 Data Overview and Preparation

We are using both the train and test datasets from kaggle competetions and combine the data for analysis .

[3]:
```
import pandas as pd
import numpy as np

# Load your train and test datasets
df_train = pd.read_csv("train_bike.csv", sep=None, engine='python')
df_test = pd.read_csv("test.csv", sep=None, engine='python')

# Identify the columns that are in the train set but not in the test set
missing_columns_in_test = [col for col in df_train.columns if col not in␣
↪df_test.columns]
```

```python
# For each missing column, add it to the test set with null values
for col in missing_columns_in_test:
    df_test[col] = None

# Combine the train and test datasets by concatenating them
df_combined = pd.concat([df_train, df_test], ignore_index=True)

# df_combined now contains the combined data with null values for missing␣
↪columns
```

[4]:
```python
import numpy as np
import pandas as pd

# Convert the 'registered' column to numeric (forcing non-numeric values to NaN)
df_combined['registered'] = pd.to_numeric(df_combined['registered'],␣
↪errors='coerce')
df_combined['casual'] = pd.to_numeric(df_combined['casual'], errors='coerce')
df_combined['count'] = pd.to_numeric(df_combined['count'], errors='coerce')

# Check for NaN values
nan_count = df_combined['registered'].isna().sum()
print(f"Number of NaN values in 'registered' column: {nan_count}")

# Check for infinity values
inf_count = np.isinf(df_combined['registered']).sum()
print(f"Number of infinity values in 'registered' column: {inf_count}")

# Check for zero values
zero_count = (df_combined['registered'] == 0).sum()
print(f"Number of zero values in 'registered' column: {zero_count}")

# Check for NaN values
nan_count = df_combined['casual'].isna().sum()
print(f"Number of NaN values in 'casual' column: {nan_count}")

# Check for infinity values
inf_count = np.isinf(df_combined['casual']).sum()
print(f"Number of infinity values in 'casual' column: {inf_count}")

# Check for zero values
zero_count = (df_combined['casual'] == 0).sum()
print(f"Number of zero values in 'casual' column: {zero_count}")

# Check for NaN values
nan_count = df_combined['count'].isna().sum()
print(f"Number of NaN values in 'count' column: {nan_count}")
```

```python
# Check for infinity values
inf_count = np.isinf(df_combined['count']).sum()
print(f"Number of infinity values in 'count' column: {inf_count}")

# Check for zero values
zero_count = (df_combined['count'] == 0).sum()
print(f"Number of zero values in 'count' column: {zero_count}")
```

```
Number of NaN values in 'registered' column: 6493
Number of infinity values in 'registered' column: 0
Number of zero values in 'registered' column: 15
Number of NaN values in 'casual' column: 6493
Number of infinity values in 'casual' column: 0
Number of zero values in 'casual' column: 986
Number of NaN values in 'count' column: 6493
Number of infinity values in 'count' column: 0
Number of zero values in 'count' column: 0
```

[5]:
```python
df_combined.shape
```

[5]:
```
(17379, 12)
```

[6]:
```python
df_combined.head()
```

[6]:
```
             datetime  season  holiday  workingday  weather  temp   atemp  \
0 2011-01-01 00:00:00       1        0           0        1  9.84  14.395
1 2011-01-01 01:00:00       1        0           0        1  9.02  13.635
2 2011-01-01 02:00:00       1        0           0        1  9.02  13.635
3 2011-01-01 03:00:00       1        0           0        1  9.84  14.395
4 2011-01-01 04:00:00       1        0           0        1  9.84  14.395

   humidity  windspeed  casual  registered  count
0        81        0.0     3.0        13.0   16.0
1        80        0.0     8.0        32.0   40.0
2        80        0.0     5.0        27.0   32.0
3        75        0.0     3.0        10.0   13.0
4        75        0.0     0.0         1.0    1.0
```

[7]:
```python
df_combined.tail()
```

[7]:
```
                 datetime  season  holiday  workingday  weather   temp  \
17374 2012-12-31 19:00:00       1        0           1        2  10.66
17375 2012-12-31 20:00:00       1        0           1        2  10.66
17376 2012-12-31 21:00:00       1        0           1        1  10.66
17377 2012-12-31 22:00:00       1        0           1        1  10.66
17378 2012-12-31 23:00:00       1        0           1        1  10.66
```

```
        atemp  humidity  windspeed  casual  registered  count
17374  12.880        60    11.0014     NaN         NaN    NaN
17375  12.880        60    11.0014     NaN         NaN    NaN
17376  12.880        60    11.0014     NaN         NaN    NaN
17377  13.635        56     8.9981     NaN         NaN    NaN
17378  13.635        65     8.9981     NaN         NaN    NaN
```

The dataset dimension considered for train datset analysis has 10886 subjects and 12 features. The dataset dimension considered for test datset analysis has 6493 subjects and 11 features(without count).After combining both datasets we were able to get more subjects equal to 17379.A moderate size dataset is considered.

6 Exploratory Data Analysis

[8]:
```
df_combined.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17379 entries, 0 to 17378
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    17379 non-null  object
 1   season      17379 non-null  int64
 2   holiday     17379 non-null  int64
 3   workingday  17379 non-null  int64
 4   weather     17379 non-null  int64
 5   temp        17379 non-null  float64
 6   atemp       17379 non-null  float64
 7   humidity    17379 non-null  int64
 8   windspeed   17379 non-null  float64
 9   casual      10886 non-null  float64
 10  registered  10886 non-null  float64
 11  count       10886 non-null  float64
dtypes: float64(6), int64(5), object(1)
memory usage: 1.6+ MB
```

[9]:
```
# Check how many None (or NaN) values are present in the 'casual',␣
 ↪'registered', and 'count' columns
missing_values = df_combined[['casual', 'registered', 'count']].isnull().sum()

# Display the result
print(missing_values)
```

```
casual        6493
registered    6493
count         6493
```

```
dtype: int64
```

After combining the dataset there are Nan values in the columns casual,registered and count . As the test dataset (df_test) does not contain the 'casual', 'registered', and 'count' columns, we combine the training and test datasets for consistency.

Missing values in the combined dataset are then imputed using the KNNImputer to ensure the model has complete data to make accurate predictions.

[10]:
```python
from sklearn.impute import KNNImputer

# Define columns to apply imputation ('casual', 'registered', 'count')
columns_to_impute = ['casual', 'registered', 'count']

# Initialize the KNNImputer
imputer = KNNImputer(n_neighbors=5)

# Apply imputation to the specified columns
df_combined[columns_to_impute] = imputer.
 ↪fit_transform(df_combined[columns_to_impute])
```

[11]:
```python
# Now check again how many None (or NaN) values are present in the 'casual',␣
 ↪'registered', and 'count' columns to make sure impuataion performed
missing_values = df_combined[['casual', 'registered', 'count']].isnull().sum()

# Display the result
print(missing_values)
```

```
casual        0
registered    0
count         0
dtype: int64
```

7 Exploratory Data Analysis (EDA)

https://www.kaggle.com/competitions/bike-sharing-demand/overview

"Dataset provides the hourly rental data for a period of two years.We will be using the data from the first two weeks of the month to predict the next day. For example considering the first two weeks data of the month we can predict the next day.The objective here is to predict the total count of bikes rented during each hour covered by the test set using the information prior to the rental period.

Data Fields

datetime: hourly date + timestamp

season: 1 = spring, 2 = summer, 3 = fall, 4 = winter

holiday: whether the day is considered a holiday workingday: whether the day is neither a weekend nor holiday

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: "feels like" temperature in Celsius

humidity: relative humidity

windspeed: wind speed

casual: number of non-registered user rentals initiated

registered: number of registered user rentals initiated

count: number of total rentals"

[12]:
```
df_combined.describe()
```

[12]:
```
             season       holiday    workingday       weather          temp  \
count  17379.000000  17379.000000  17379.000000  17379.000000  17379.000000
mean       2.501640      0.028770      0.682721      1.425283     20.376474
std        1.106918      0.167165      0.465431      0.639357      7.894801
min        1.000000      0.000000      0.000000      1.000000      0.820000
25%        2.000000      0.000000      0.000000      1.000000     13.940000
50%        3.000000      0.000000      1.000000      1.000000     20.500000
75%        3.000000      0.000000      1.000000      2.000000     27.060000
max        4.000000      1.000000      1.000000      4.000000     41.000000

              atemp      humidity     windspeed        casual    registered  \
count  17379.000000  17379.000000  17379.000000  17379.000000  17379.000000
mean      23.788755     62.722884     12.736540     36.021955    155.552177
std        8.592511     19.292983      8.196795     39.540385    119.537321
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%       16.665000     48.000000      7.001500     10.000000     85.000000
50%       24.240000     63.000000     12.998000     36.021955    155.552177
75%       31.060000     78.000000     16.997900     36.021955    155.552177
max       50.000000    100.000000     56.996900    367.000000    886.000000

              count
count  17379.000000
mean     191.574132
std      143.363753
min        1.000000
25%      101.000000
```

```
50%        191.574132
75%        192.000000
max        977.000000
```

Inference:

season, holiday,workingday,weather are the categorical features.

temparature,atemp,humidity,windspeed,casual,registered are the numerical features.

The dependent feature is count (prediction variable in a model i.e ,y)

[13]:
```
# Let's check for NULLs
df_combined.isna().sum()
```

[13]:
```
datetime      0
season        0
holiday       0
workingday    0
weather       0
temp          0
atemp         0
humidity      0
windspeed     0
casual        0
registered    0
count         0
dtype: int64
```

No null values found in any of the feature values.

[14]:
```
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

[15]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

[16]:
```
# First, we have to convert the type of this column
df_combined["datetime"] = pd.to_datetime(df_combined["datetime"])

# Then we can create some more granular features related to time
df_combined["hour"] = df_combined["datetime"].dt.hour
df_combined["day_of_week"] = df_combined["datetime"].dt.dayofweek
df_combined["month"] = df_combined["datetime"].dt.month
df_combined["day_of_month"] = df_combined["datetime"].dt.day
```

[17]:
```
df_combined.head()
```

[17]:
```
                  datetime  season  holiday  workingday  weather  temp   atemp  \
0 2011-01-01 00:00:00       1        0           0        1  9.84  14.395
1 2011-01-01 01:00:00       1        0           0        1  9.02  13.635
2 2011-01-01 02:00:00       1        0           0        1  9.02  13.635
3 2011-01-01 03:00:00       1        0           0        1  9.84  14.395
4 2011-01-01 04:00:00       1        0           0        1  9.84  14.395

   humidity  windspeed  casual  registered  count  hour  day_of_week  month  \
0        81        0.0     3.0        13.0   16.0     0            5      1
1        80        0.0     8.0        32.0   40.0     1            5      1
2        80        0.0     5.0        27.0   32.0     2            5      1
3        75        0.0     3.0        10.0   13.0     3            5      1
4        75        0.0     0.0         1.0    1.0     4            5      1

   day_of_month
0             1
1             1
2             1
3             1
4             1
```

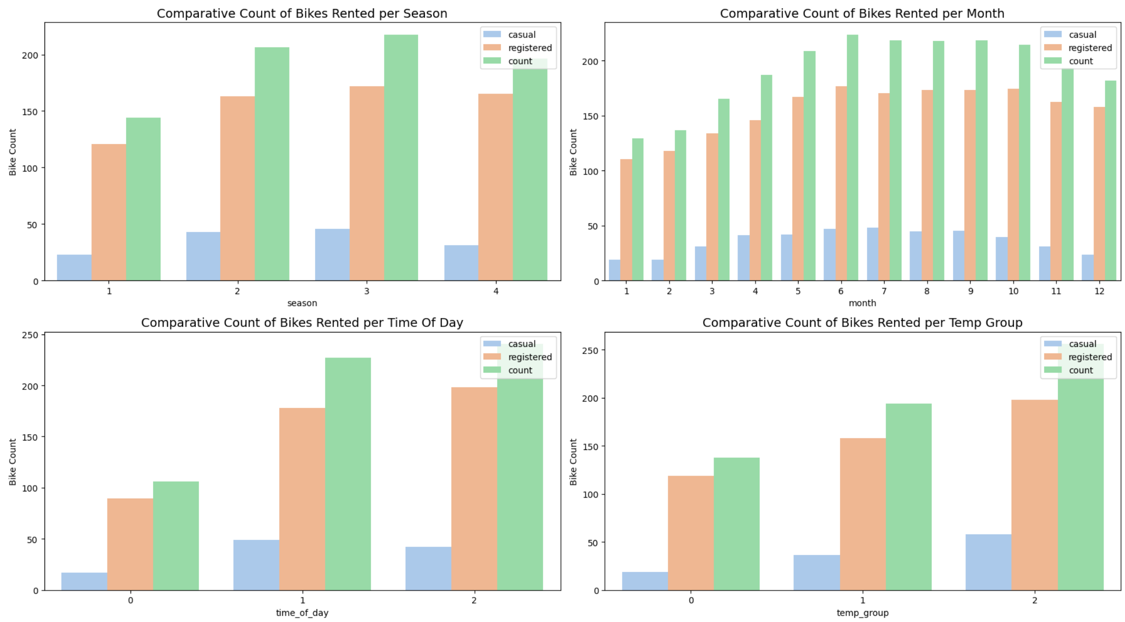
[18]:
```
df_combined.tail()
```

[18]:
```
                      datetime  season  holiday  workingday  weather   temp  \
17374 2012-12-31 19:00:00       1        0           1        2  10.66
17375 2012-12-31 20:00:00       1        0           1        2  10.66
17376 2012-12-31 21:00:00       1        0           1        1  10.66
17377 2012-12-31 22:00:00       1        0           1        1  10.66
17378 2012-12-31 23:00:00       1        0           1        1  10.66

        atemp  humidity  windspeed     casual  registered       count  hour  \
17374  12.880        60    11.0014  36.021955  155.552177  191.574132    19
17375  12.880        60    11.0014  36.021955  155.552177  191.574132    20
17376  12.880        60    11.0014  36.021955  155.552177  191.574132    21
17377  13.635        56     8.9981  36.021955  155.552177  191.574132    22
17378  13.635        65     8.9981  36.021955  155.552177  191.574132    23

       day_of_week  month  day_of_month
17374            0     12            31
17375            0     12            31
17376            0     12            31
17377            0     12            31
17378            0     12            31
```

[19]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

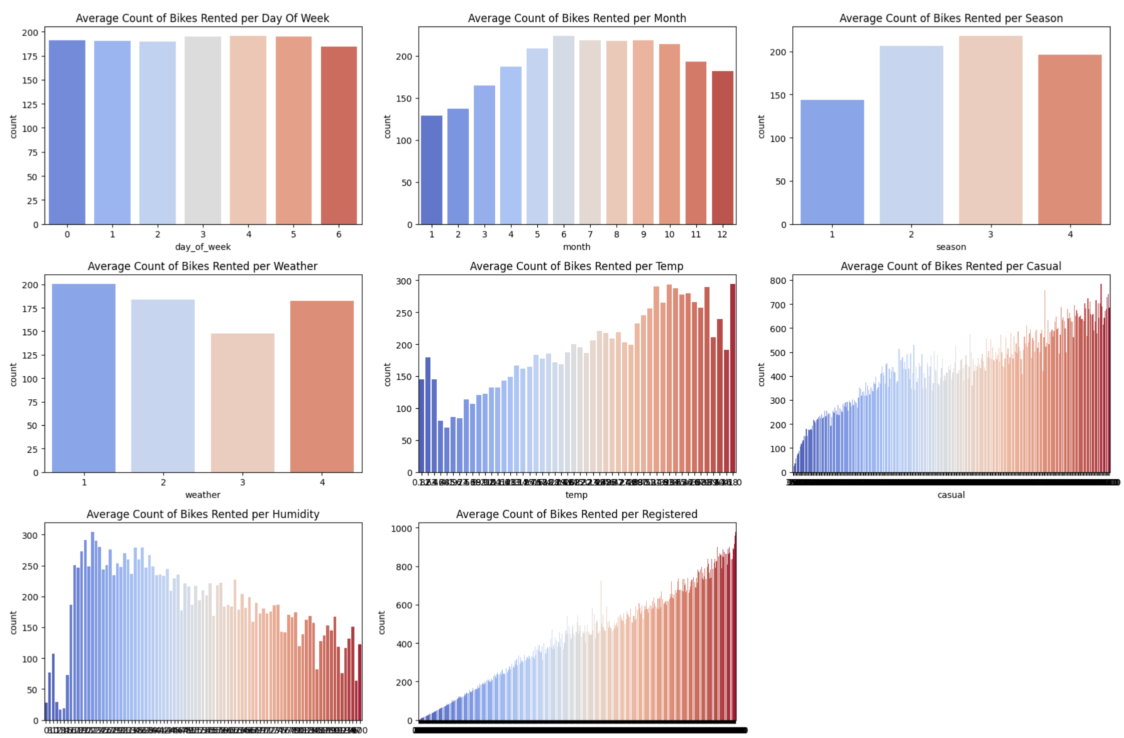
```
import warnings
warnings.filterwarnings("ignore", category=UserWarning)

def plot_bike_counts_subplots(features):
    # Create a figure with subplots for each feature (use a 2x3 grid for 6␣
 ↪features)
    fig, axes = plt.subplots(2, 2, figsize=(18, 10))  # 2x2 grid of plots
    axes = axes.flatten()  # Flatten the axes array for easier access

    # Soft pastel color palette
    soft_palette = sns.color_palette("pastel")

    # Iterate over the features and plot each one
    for i, feature in enumerate(features):
        # For 'temp' and 'atemp', bin the data into ranges (Low, Medium, High␣
 ↪as 0, 1, 2) and store them as 'temp_group' and 'atemp_group'
        if feature == 'temp':
            df_combined['temp_group'] = pd.cut(df_combined['temp'], bins=3,␣
 ↪labels=[0, 1, 2])
            grouped_data = df_combined.groupby('temp_group')[['casual',␣
 ↪'registered', 'count']].mean().reset_index()
            feature = 'temp_group'

        # For 'hour', group into 3 categories: 0 for Early Morning, 1 for␣
 ↪Morning/Afternoon, 2 for Evening
        elif feature == 'hour':
            df_combined['time_of_day'] = pd.cut(df_combined['hour'], bins=[0,␣
 ↪8, 16, 24],
                                                          labels=[0, 1, 2],␣
 ↪right=False)
            grouped_data = df_combined.groupby('time_of_day')[['casual',␣
 ↪'registered', 'count']].mean().reset_index()
            feature = 'time_of_day'

        else:
            # Group by the feature and calculate the mean for 'casual',␣
 ↪'registered', and 'count'
            grouped_data = df_combined.groupby(feature)[['casual',␣
 ↪'registered', 'count']].mean().reset_index()

        # Melt the data for easier plotting
        grouped_data_melted = grouped_data.melt(id_vars=feature,␣
 ↪value_vars=['casual', 'registered', 'count'],
                                                 var_name='User Type',␣
 ↪value_name='Bike Count')
```

```
        # Plot the comparative bar chart on the respective subplot axis
        sns.barplot(x=feature, y='Bike Count', data=grouped_data_melted,␣
↪hue='User Type', palette=soft_palette, ax=axes[i])

        # Format the title
        title_feature = feature.replace('_', ' ').title()
        axes[i].set_title(f'Comparative Count of Bikes Rented per␣
↪{title_feature}', fontsize=14)
        axes[i].legend(loc='upper right')

    # Adjust layout to prevent overlap
    plt.tight_layout()
    plt.show()

# Example usage: 6 plots in one image
features = ['season', 'month', 'hour', 'temp']
plot_bike_counts_subplots(features)
```



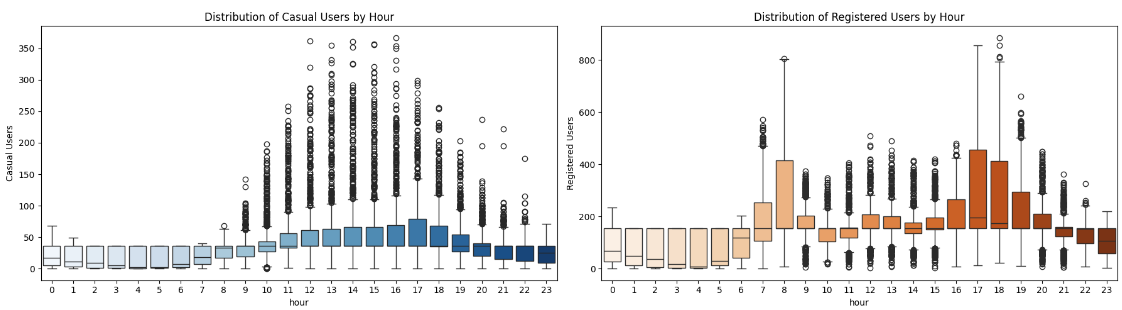
[20]:
```
import matplotlib.pyplot as plt
import seaborn as sns

def plot_bike_counts_grid(features: list, rows: int = 3, cols: int = 3):
    fig, axes = plt.subplots(rows, cols, figsize=(18, 12))
    axes = axes.flatten()

    # Iterate over the features and plot each one
```

```
    for i, feature in enumerate(features):
        data = df_combined.groupby(feature)['count'].mean().reset_index()
        sns.barplot(ax=axes[i], x=feature, y='count', data=data,␣
↪palette='coolwarm')

        # Format the title based on the feature name
        title_feature = feature.replace('_', ' ').title()
        axes[i].set_title(f'Average Count of Bikes Rented per {title_feature}')

    # Hide any unused subplots if there are fewer features than subplots
    for j in range(i+1, rows * cols):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()

# Example usage:
plot_bike_counts_grid(['day_of_week', 'month', 'season',␣
↪'weather','temp','casual','humidity','registered'])
```



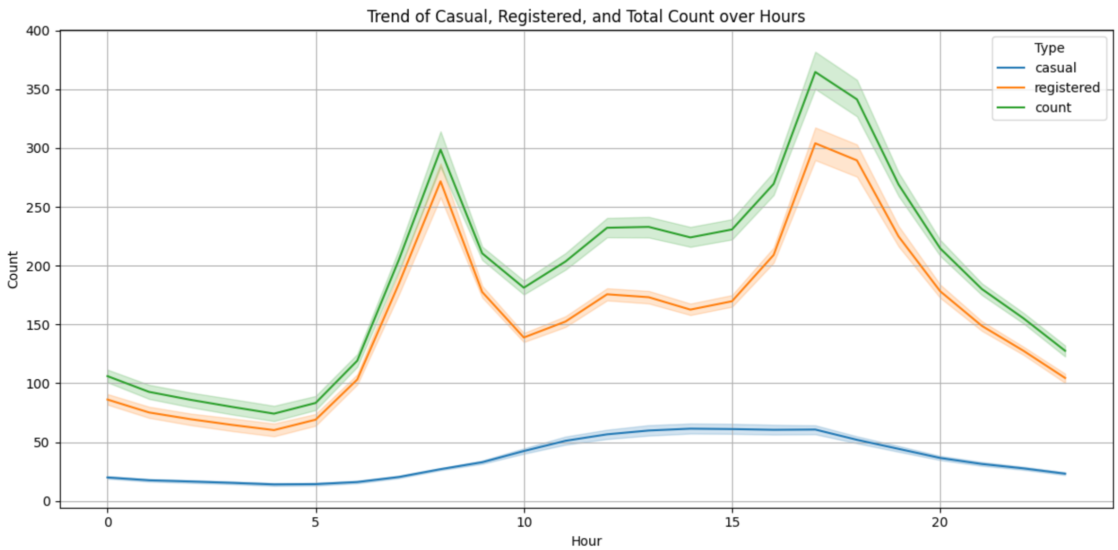
[21]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
# Define a color palette for the boxplots
casual_palette = sns.color_palette("Blues", len(df_combined['hour'].unique()))
registered_palette = sns.color_palette("Oranges", len(df_combined['hour'].unique()))
# Create the subplots
fig, axs = plt.subplots(1, 2, figsize=(18, 5), sharex=False, sharey=False)
# Plot for casual users with custom color palette
sns.boxplot(x='hour', y='casual', data=df_combined, ax=axs[0], palette=casual_palette)
axs[0].set_ylabel('Casual Users')
axs[0].set_title('Distribution of Casual Users by Hour')
# Plot for registered users with custom color palette
sns.boxplot(x='hour', y='registered', data=df_combined, ax=axs[1], palette=registered_palette)
axs[1].set_ylabel('Registered Users')
axs[1].set_title('Distribution of Registered Users by Hour')
plt.tight_layout()
plt.show()
```



[24]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Set up the figure for a single plot
fig, ax = plt.subplots(figsize=(12, 6))  # Single plot

# Plot: Trend of Casual, Registered, and Total Count over Hours
# Melt the dataframe to long format for easier plotting with seaborn
df_melted_hour = df_combined.melt(id_vars=['hour'], value_vars=['casual', 'registered', 'count'],
                                  var_name='Type', value_name='Count')

# Seaborn line plot for hours
sns.lineplot(data=df_melted_hour, x='hour', y='Count', hue='Type', ax=ax)
ax.set_title('Trend of Casual, Registered, and Total Count over Hours')
```

```
ax.set_xlabel('Hour')
ax.set_ylabel('Count')
ax.grid(True)

# Show the plot
plt.tight_layout()
plt.show()
```



The graph illustrates hourly bike rental patterns, revealing distinct trends for registered and casual users:

Registered users dominate the rental activity, closely mirroring the total rental curve. Two prominant peaks , one in the morning (around 8 AM) and another in the late afternoon (around 5 PM).A significant drop during nighttime hours.

Casual rentals exhibit a much flatter pattern throughout the day. Consistently lower volume compared to registered users. Slight increase during midday hours. Less pronounced peaks and valleys.

Key Insights:

Registered users likely drive commuter traffic, given the morning and evening peaks. Casual users show more steady, possibly leisure-oriented usage. The system experiences highest demand during typical rush hours. Night time sees the lowest rental activity for both user types. This data suggests the need for targeted strategies to optimize bike availability and potentially increase casual user engagement during off-peak hours.

8 EDA Observations:

1. Temporal patterns:

There is a clear cyclical trend in bike usage across different hours of the day, with peaks during morning (around 8 AM) and evening (around 5-6 PM), which correspond to commuting hours. This suggests that bike demand is closely tied to work schedules.

Weekday vs. Weekend trends reveal that demand is higher on weekdays, especially for registered users (likely commuters), while casual users are more prominent on weekends. This indicates that bike-sharing services might be used more for leisure on weekends and for commuting on weekdays.

Bike rentals increase during the summer months (June, July, August) and drop in the winter, indicating seasonal dependence. This could be due to favorable weather conditions for cycling in the summer.

2. Casual vs. Registered Users:

The majority of bike rentals are made by registered users, with their demand following a more structured daily pattern, peaking during typical commuting hours (morning and evening). This suggests that registered users predominantly use the service for commuting purposes.

Casual users, on the other hand, tend to use bikes more on weekends and during the day (outside commuting hours). This suggests a more recreational usage pattern for this group.

There is a clear opportunity to convert casual users into registered users, especially by targeting them with weekend-specific promotions and leisure-time packages.

3. Seasonal Influence:

Spring and summer seasons show higher bike demand compared to fall and winter. This could be due to favorable outdoor conditions, such as more sunlight and milder temperatures during these seasons.

Casual users seem more affected by the seasons, with their activity significantly increasing in summer, while registered users maintain a more consistent demand across seasons (albeit still higher in summer).

4. Weather and Demand:

As expected, temperature shows a positive correlation with bike rentals, with higher demand observed on warmer days. However, extremely high temperatures may lead to a slight dip in demand, indicating that there is an optimal temperature range for biking.

There is a negative correlation between humidity and bike demand, meaning bike rentals tend to decrease as humidity increases. Higher humidity might make riding less comfortable for users.

Clear or partly cloudy weather is associated with higher bike usage, while bike demand drops on rainy or snowy days. This observation highlights the importance of weather as a key factor in predicting bike demand. Highest count of bikes are rented when the weather is Clear, Few clouds, Partly cloudy, Partly cloudy and Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist

9 Summary:

Peak demand occurs during typical commuting hours on weekdays (for registered users) and in the afternoons on weekends (for casual users).

Weather conditions play a significant role, with temperature and clear skies positively influencing demand, while rain and humidity have a negative effect.

Seasonality affects casual users more than registered users, with demand peaking in summer and dropping in winter.

Registered users have more predictable patterns, making them the primary target for operational efficiency, while casual users present opportunities for conversion into registered users through targeted offerings.

These observations provide a solid foundation for predictive modeling and business strategies, such as optimizing bike placements, adjusting to seasonal demand, and creating tailored promotions to convert casual users into registered customers.

10 Feature Engineering

[25]:

```
import math
import numpy as np

# Create the interaction feature 'temp_hour_interaction' by multiplying 'temp'␣
↪and 'hour'
df_combined['temp_hour_interaction'] = df_combined['temp'] * df_combined['hour']

# Create the interaction feature 'humidity_hour_interaction' by multiplying␣
↪'humidity' and 'hour'
df_combined['humidity_hour_interaction'] = df_combined['humidity'] *␣
↪df_combined['hour']

# Create the interaction feature 'windspeed_hour_interaction' by multiplying␣
↪'windspeed' and 'hour'
df_combined['windspeed_hour_interaction'] = df_combined['windspeed'] *␣
↪df_combined['hour']

# Extract the day of the week from the 'datetime' column (Monday=0, Sunday=6)
df_combined['day_of_week'] = df_combined['datetime'].dt.dayofweek

# Create sine and cosine transformations for 'day_of_week', 'hour', and 'month'

# Day of the week (7 unique values, so use 7 in the formula)
df_combined['sin_day_of_week'] = np.sin(2 * np.pi * df_combined['day_of_week'] /
↪ 7)
df_combined['cos_day_of_week'] = np.cos(2 * np.pi * df_combined['day_of_week'] /
↪ 7)

# Hour of the day (24 unique values, so use 24 in the formula)
df_combined['sin_hour'] = np.sin(2 * np.pi * df_combined['hour'] / 24)
df_combined['cos_hour'] = np.cos(2 * np.pi * df_combined['hour'] / 24)
```

```python
# Month of the year (12 unique values, so use 12 in the formula)
df_combined['sin_month'] = np.sin(2 * np.pi * df_combined['month'] / 12)
df_combined['cos_month'] = np.cos(2 * np.pi * df_combined['month'] / 12)

# Create a new feature 'temp_diff' as the difference between 'temp' and 'atemp'
df_combined['temp_diff'] = df_combined['temp'] - df_combined['atemp']

# Create a new feature 'workingday_weather_interaction' by multiplying␣
↪'workingday' and 'weather'
df_combined['workingday_weather_interaction'] = df_combined['workingday'] *␣
↪df_combined['weather']

# Extract the unique holidays from 'datetime' where holiday is marked as 1, and␣
↪remove NaT values
holidays = df_combined[df_combined['holiday'] == 1]['datetime'].dropna().dt.
↪date.unique()

# Define a function to calculate proximity to the nearest holiday
def calculate_holiday_proximity(current_date, holidays):
    # Ensure current_date is valid and not NaT
    if pd.isnull(current_date):
        return None  # Return None if current_date is NaT
    current_date = current_date.date()  # Extract the date part
    # Calculate the difference (in days) between the given date and all␣
↪holidays, take the minimum
    return min(abs((current_date - holiday).days) for holiday in holidays)

# Apply the function to calculate proximity to holidays for each row
df_combined['holiday_proximity'] = df_combined['datetime'].apply(lambda x:␣
↪calculate_holiday_proximity(x, holidays))

# Ensure that the 'casual' column is numeric (if necessary, convert it)
df_combined['casual'] = pd.to_numeric(df_combined['casual'], errors='coerce')

# Create the lag feature 'lag_casual_1', which is the 'casual' value from the␣
↪previous row
df_combined['lag_casual_1'] = df_combined['casual'].shift(1)

# Ensure that the 'registered' column is numeric (if necessary, convert it)
df_combined['registered'] = pd.to_numeric(df_combined['registered'],␣
↪errors='coerce')

# Create the lag feature 'lag_registered_1', which is the 'registered' value␣
↪from the previous row
df_combined['lag_registered_1'] = df_combined['registered'].shift(1)
```

```
# Extract the quarter from the 'datetime' column
df_combined['quarter'] = df_combined['datetime'].dt.quarter

## Apply log transformation (log(1 + casual)) to handle zeros
df_combined['log_casual'] = np.log1p(df_combined['casual'])

# Apply log transformation (log(1 + registered)) to handle zeros
df_combined['log_registered'] = np.log1p(df_combined['registered'])

## Calculate the conversion ratio (Registered Riders / (Casual Riders + 1))
df_combined['conversion_ratio'] = df_combined['log_registered'] /
↪(df_combined['log_casual'] + 1)
```

[26]:
```
df_combined.shape
```

[26]: (17379, 36)

[27]:
```
zero_count = (df_combined['conversion_ratio'] >= 0).sum()

print(f"Number of zero values in 'casual' column: {zero_count}")
```

```
Number of zero values in 'casual' column: 17379
```

[28]:
```
nan_count = df_combined['log_casual'].isna().sum()
print(f"Number of NaN values in 'registered' column: {nan_count}")

# Check for infinity values
inf_count = np.isinf(df_combined['log_casual']).sum()
print(f"Number of infinity values in 'registered' column: {inf_count}")

# Check for zero values
zero_count = (df_combined['log_casual'] == 0).sum()
print(f"Number of zero values in 'registered' column: {zero_count}")
```

```
Number of NaN values in 'registered' column: 0
Number of infinity values in 'registered' column: 0
Number of zero values in 'registered' column: 986
```

After extracting few more features from the existing features we were able to get reasonable amount of features for modelling

11 Data Cleaning

[29]:
```
df_combined.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17379 entries, 0 to 17378
```

```
Data columns (total 36 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   datetime                        17379 non-null  datetime64[ns]
 1   season                          17379 non-null  int64
 2   holiday                         17379 non-null  int64
 3   workingday                      17379 non-null  int64
 4   weather                         17379 non-null  int64
 5   temp                            17379 non-null  float64
 6   atemp                           17379 non-null  float64
 7   humidity                        17379 non-null  int64
 8   windspeed                       17379 non-null  float64
 9   casual                          17379 non-null  float64
 10  registered                      17379 non-null  float64
 11  count                           17379 non-null  float64
 12  hour                            17379 non-null  int32
 13  day_of_week                     17379 non-null  int32
 14  month                           17379 non-null  int32
 15  day_of_month                    17379 non-null  int32
 16  time_of_day                     17379 non-null  category
 17  temp_group                      17379 non-null  category
 18  temp_hour_interaction           17379 non-null  float64
 19  humidity_hour_interaction       17379 non-null  int64
 20  windspeed_hour_interaction      17379 non-null  float64
 21  sin_day_of_week                 17379 non-null  float64
 22  cos_day_of_week                 17379 non-null  float64
 23  sin_hour                        17379 non-null  float64
 24  cos_hour                        17379 non-null  float64
 25  sin_month                       17379 non-null  float64
 26  cos_month                       17379 non-null  float64
 27  temp_diff                       17379 non-null  float64
 28  workingday_weather_interaction  17379 non-null  int64
 29  holiday_proximity               17379 non-null  int64
 30  lag_casual_1                    17378 non-null  float64
 31  lag_registered_1                17378 non-null  float64
 32  quarter                         17379 non-null  int32
 33  log_casual                      17379 non-null  float64
 34  log_registered                  17379 non-null  float64
 35  conversion_ratio                17379 non-null  float64
dtypes: category(2), datetime64[ns](1), float64(20), int32(5), int64(8)
memory usage: 4.2 MB
```

[30]:

```python
# Find which columns have NaN values
columns_with_nan = df_combined.columns[df_combined.isnull().any()]

# Display the columns with NaN values
print("Columns with NaN values:", columns_with_nan)
```

```
Columns with NaN values: Index(['lag_casual_1', 'lag_registered_1'],
dtype='object')
```

[31]:

```python
from sklearn.impute import KNNImputer
import pandas as pd

# Define the columns to impute
cols_to_impute = ['lag_casual_1', 'lag_registered_1']

# Separate the columns to impute
df_to_impute = df_combined[cols_to_impute]

# Initialize the KNNImputer with desired parameters (e.g., 5 neighbors)
imputer = KNNImputer(n_neighbors=5)

# Perform the KNN imputation
df_imputed = imputer.fit_transform(df_to_impute)

# Convert the imputed result back to a DataFrame with the original column names
df_imputed = pd.DataFrame(df_imputed, columns=cols_to_impute)

# Replace the original columns in the DataFrame with the imputed values
df_combined[cols_to_impute] = df_imputed

# Display the updated DataFrame with imputed columns
df_combined.head()
```

[31]:

```
             datetime  season  holiday  workingday  weather  temp   atemp  \
0 2011-01-01 00:00:00       1        0           0        1  9.84  14.395
1 2011-01-01 01:00:00       1        0           0        1  9.02  13.635
2 2011-01-01 02:00:00       1        0           0        1  9.02  13.635
3 2011-01-01 03:00:00       1        0           0        1  9.84  14.395
4 2011-01-01 04:00:00       1        0           0        1  9.84  14.395

   humidity  windspeed  casual  …  cos_month  temp_diff  \
0        81        0.0     3.0  …   0.866025     -4.555
1        80        0.0     8.0  …   0.866025     -4.615
2        80        0.0     5.0  …   0.866025     -4.615
3        75        0.0     3.0  …   0.866025     -4.555
4        75        0.0     0.0  …   0.866025     -4.555

   workingday_weather_interaction  holiday_proximity  lag_casual_1  \
0                               0                 16     36.021955
1                               0                 16      3.000000
2                               0                 16      8.000000
3                               0                 16      5.000000
4                               0                 16      3.000000
```

```
   lag_registered_1 quarter log_casual  log_registered  conversion_ratio
0        155.552177       1   1.386294        2.639057          1.105923
1         13.000000       1   2.197225        3.496508          1.093607
2         32.000000       1   1.791759        3.332205          1.193586
3         27.000000       1   1.386294        2.397895          1.004861
4         10.000000       1   0.000000        0.693147          0.693147

[5 rows x 36 columns]
```

[32]:
```
# Find which columns have NaN values
columns_with_nan = df_combined.columns[df_combined.isnull().any()]

# Display the columns with NaN values
print("Columns with NaN values:", columns_with_nan)
```

```
Columns with NaN values: Index([], dtype='object')
```

[33]:
```
def remove_outliers_iqr(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 4.5 * IQR
    upper_bound = Q3 + 4.5 * IQR

    # Identify the outliers
    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)]

    # Filter the data to remove outliers
    df_filtered = df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]

    print(f"Number of outliers removed: {len(outliers)}")

    return df_filtered

# Removing outliers from the 'registered' column
df_cleaned = remove_outliers_iqr(df_combined, 'registered')

df_cleaned.shape
```

```
Number of outliers removed: 505
```

[33]:
```
(16874, 36)
```

After performing imputation we were able to clean the data well for further processing.

12 Correlation

The correlation matrix is a powerful tool used to measure the relationships between independent variables in a dataset. It helps identify how strongly pairs of variables are linearly related to each other.

Correlation values range from -1 to 1:

A value close to 1 indicates a strong positive correlation, meaning as one variable increases, the other tends to increase as well. A value close to -1 indicates a strong negative correlation, where one variable increases as the other decreases. A value around 0 indicates little to no linear relationship between the variables.

Importance of the Correlation Matrix:

Identify Multicollinearity: If two or more independent variables are highly correlated, it can lead to multicollinearity, which can negatively impact the model's performance. High multicollinearity makes it difficult to determine the individual effect of each variable on the dependent variable.

Feature Selection: By analyzing the correlation matrix, we can detect variables that are redundant or too closely related to others. This helps in deciding which features to retain and which to drop for building a simpler and more efficient model.

Insights into Relationships: Understanding correlations helps in interpreting the underlying relationships in the data. For example, we can see how features like temperature and atemp (feels-like temperature) are closely related, which allows us to make more informed decisions on feature engineering and selection.

By using the correlation matrix effectively, we can reduce redundancy in the dataset, improve model performance, and gain insights into the relationships between variables, making it a critical step in the data preprocessing phase.

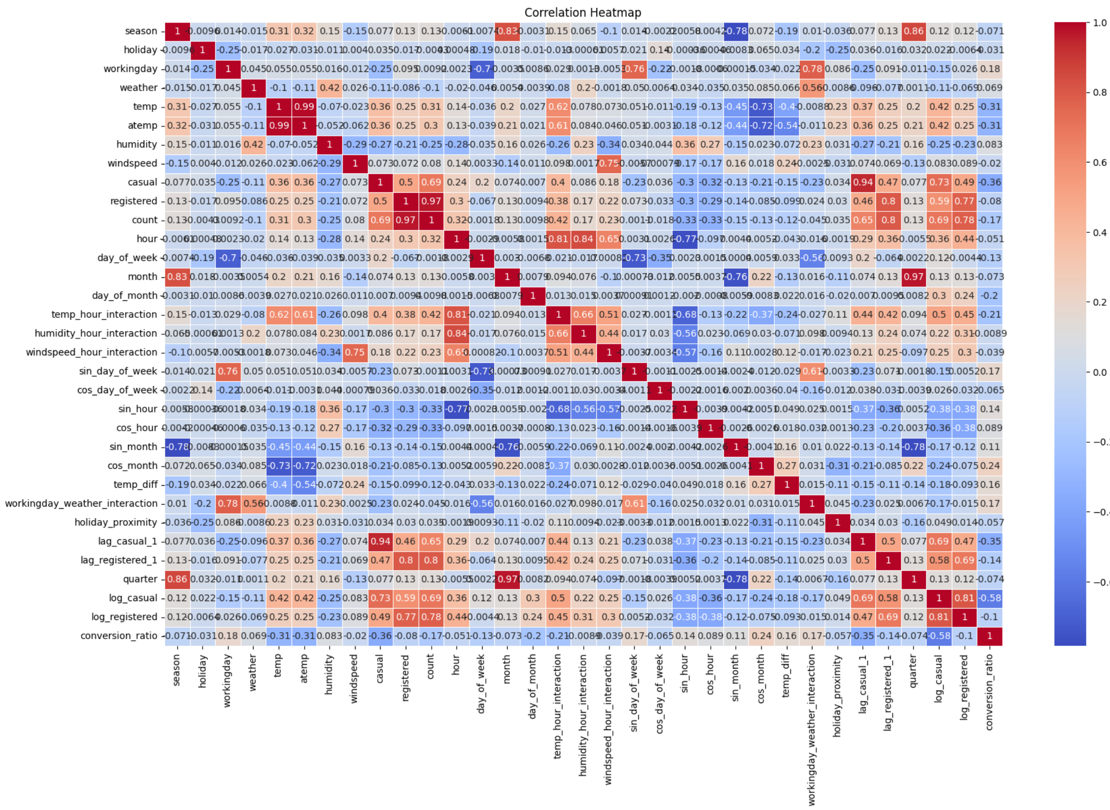
Dropping highly correlated features is important to:

Avoid Redundancy: Highly correlated features carry similar information, so keeping both doesn't add value to the model.

Prevent Multicollinearity: Multicollinearity makes model coefficients unstable, increases variance, and makes it hard to interpret individual feature importance.

Reduce Overfitting: Removing redundant features helps the model generalize better to unseen data.

[34]:
```
plt.figure(figsize=(20,12))
sns.heatmap(df_combined.corr(numeric_only=True), annot=True, cmap='coolwarm',␣
 ↪linewidths=.5)
plt.title('Correlation Heatmap')
plt.show()
```



Why Remove Features with Perfect Correlation to the Target Variable?

1. Redundancy: A feature that is perfectly correlated with the target adds no new information to the model because it directly predicts the target variable. Including such a feature may cause overfitting.
2. Data Leakage: If a feature is perfectly correlated with the target, it may indicate data leakage—where information from the target is already present in the features, which makes the model overly optimistic and not generalizable to unseen data.
3. Model Simplification: Removing such features simplifies the model, making it less prone to overfitting and easier to interpret.

[35]:
```
df_combined = df_combined.drop(['datetime', 'atemp', 'registered', 'month',␣
↪'casual'], axis=1)
```

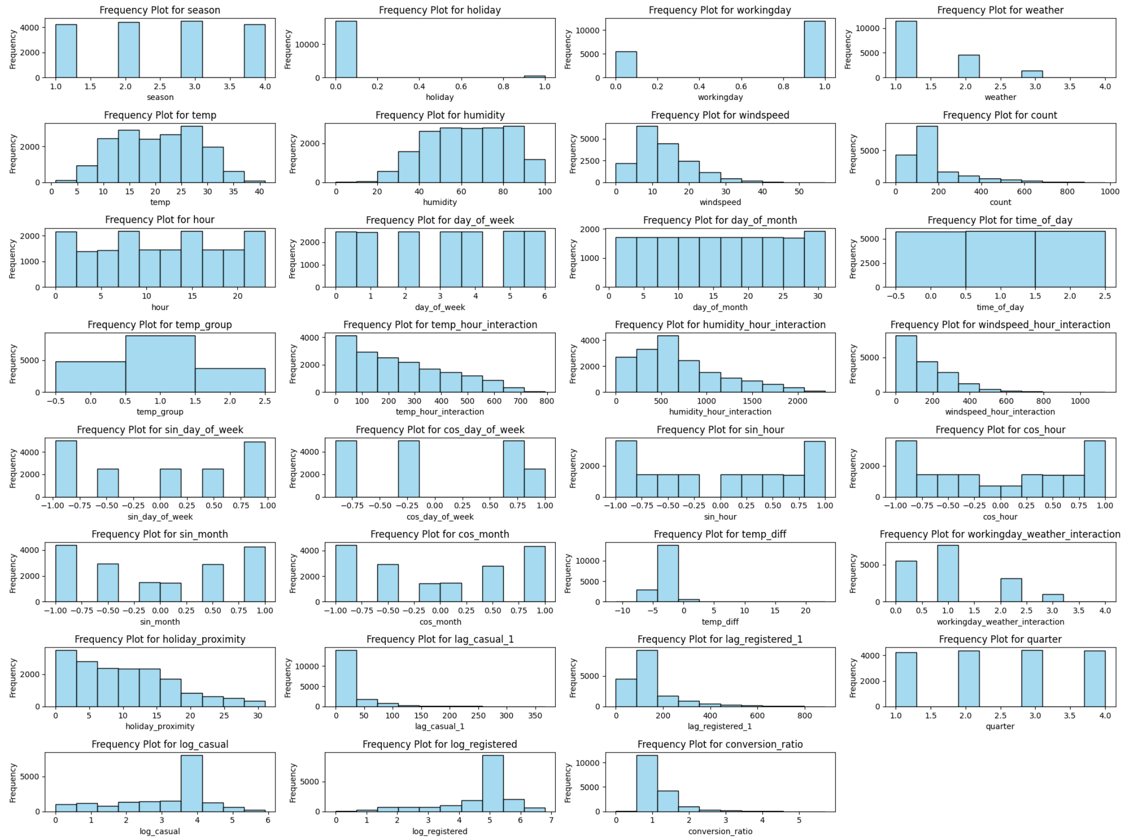
After dropping the features which are correlated each other .check for categorical features which also can be removed after extracting numerical information from those.

[36]:
```
df_combined.describe()
```

[36]:
```
             season       holiday    workingday       weather          temp  \
count  17379.000000  17379.000000  17379.000000  17379.000000  17379.000000
mean       2.501640      0.028770      0.682721      1.425283     20.376474
std        1.106918      0.167165      0.465431      0.639357      7.894801
```

```
min        1.000000      0.000000      0.000000      1.000000      0.820000
25%        2.000000      0.000000      0.000000      1.000000     13.940000
50%        3.000000      0.000000      1.000000      1.000000     20.500000
75%        3.000000      0.000000      1.000000      2.000000     27.060000
max        4.000000      1.000000      1.000000      4.000000     41.000000

           humidity     windspeed         count          hour   day_of_week  \
count  17379.000000  17379.000000  17379.000000  17379.000000  17379.000000
mean      62.722884     12.736540    191.574132     11.546752      3.011451
std       19.292983      8.196795    143.363753      6.914405      2.001966
min        0.000000      0.000000      1.000000      0.000000      0.000000
25%       48.000000      7.001500    101.000000      6.000000      1.000000
50%       63.000000     12.998000    191.574132     12.000000      3.000000
75%       78.000000     16.997900    192.000000     18.000000      5.000000
max      100.000000     56.996900    977.000000     23.000000      6.000000

       …     cos_month     temp_diff  workingday_weather_interaction  \
count  …  1.737900e+04  17379.000000                    17379.000000
mean   … -6.459681e-03     -3.412281                        0.986363
std    …  7.087640e-01      1.469472                        0.860903
min    … -1.000000e+00    -11.075000                        0.000000
25%    … -8.660254e-01     -4.055000                        0.000000
50%    … -1.836970e-16     -3.680000                        1.000000
75%    …  8.660254e-01     -2.680000                        1.000000
max    …  1.000000e+00     23.140000                        4.000000

       holiday_proximity  lag_casual_1  lag_registered_1       quarter  \
count       17379.000000  17379.000000      17379.000000  17379.000000
mean           10.408539     36.021955        155.552177      2.512055
std             7.195123     39.540385        119.537321      1.114108
min             0.000000      0.000000          0.000000      1.000000
25%             4.000000     10.000000         85.000000      2.000000
50%            10.000000     36.021955        155.552177      3.000000
75%            15.000000     36.021955        155.552177      4.000000
max            31.000000    367.000000        886.000000      4.000000

         log_casual  log_registered  conversion_ratio
count  17379.000000    17379.000000      17379.000000
mean       3.048343        4.641467          1.221635
std        1.257675        1.152306          0.414281
min        0.000000        0.000000          0.000000
25%        2.397895        4.454347          1.095796
50%        3.611511        5.053389          1.095821
75%        3.611511        5.053389          1.241842
max        5.908083        6.787845          5.707110

[8 rows x 29 columns]
```

[37]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Features to generate frequency plots and tables for
features = df_combined.columns.tolist()

# Set up the number of rows and columns for subplots
n_cols = 4
n_rows = (len(features) + n_cols - 1) // n_cols  # Calculate rows required for given columns

# Create subplots
fig, axes = plt.subplots(n_rows, n_cols, figsize=(20, 15))
axes = axes.flatten()  # Flatten the axes array for easier access

# Plot each feature in a subplot
for i, feature in enumerate(features):
    sns.histplot(df_combined[feature], kde=False, ax=axes[i], bins=10, color='#87CEEB')
    axes[i].set_title(f"Frequency Plot for {feature}")
    axes[i].set_xlabel(feature)
    axes[i].set_ylabel("Frequency")

# Remove any empty subplots
for i in range(len(features), len(axes)):
    fig.delaxes(axes[i])

# Adjust the layout
plt.tight_layout()
plt.show()
```



13 Modelling

Splitting the data into training and test sets is a crucial step in building any machine learning model. The purpose of this exercise is to evaluate the model's ability to generalize to unseen data. Here's why this step is important:

Importance of Train-Test Split:

Model Training:

The training set is used to train the model. During this phase, the model learns the relationships between the input features (independent variables) and the target variable (dependent variable). By learning from this data, the model adjusts its parameters to minimize prediction error.

Model Evaluation:

The test set is used to evaluate the model's performance. The test data acts as new, unseen data, allowing us to measure how well the model generalizes beyond the data it was trained on. A good model should perform well on both the training and test sets. If it performs well on the training set but poorly on the test set, the model may be overfitting.

Avoiding Overfitting:

Overfitting occurs when a model becomes too specialized in the training data, capturing noise or irrelevant patterns that do not generalize well to new data. By having a test set, we can detect overfitting early and take corrective measures, such as using regularization or simplifying the model.

Performance Metrics:

After training the model on the training set, we use the test set to calculate performance metrics such as mean squared error (MSE), R-squared (R^2), and others. These metrics give us an objective measure of how well the model performs in a real-world scenario.

Typical Split Ratio:

The data is typically split in a 80/20 or 70/30 ratio, where:

80% (or 70%) of the data is used for training the model. 20% (or 30%) of the data is set aside for testing. This split ensures that the model gets enough data to learn effectively, while leaving enough data for a robust evaluation of its performance.

[38]:
```
#importing necessary modules for Deploying models and Evaluating them
from sklearn.metrics import mean_absolute_error, r2_score, mean_squared_error,␣
↪max_error
from sklearn.linear_model import  LinearRegression, Lasso, Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR, LinearSVR
from sklearn.neighbors import KNeighborsRegressor                              ␣
↪                                                                              ␣
↪                                                                              ␣
↪                                                       #mahinisawesomemate
from sklearn.ensemble import AdaBoostRegressor, BaggingRegressor,␣
↪GradientBoostingRegressor, RandomForestRegressor, VotingRegressor,␣
↪StackingRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
```

Importance of StandardScaler and PCA StandardScaler StandardScaler is a preprocessing technique used to standardize features by removing the mean and scaling them to unit variance. It ensures that each feature contributes equally to the model and prevents bias toward features with larger values. Here's why it's important:

Normalization of Features:

Different features in the dataset may have varying scales. For example, temperature may be measured in degrees, while wind speed is measured in meters per second. Without scaling, features with larger values might dominate the learning process, leading to biased predictions.

Model Performance:

Many machine learning algorithms, especially those that use distance metrics (like k-nearest neighbors or support vector machines), perform better when features are on a similar scale. StandardScaler ensures that all features contribute equally to the model.

Stability of Coefficients:

Scaling improves the stability of the model coefficients, especially in models like linear regression and logistic regression, where coefficients represent the influence of each feature on the target variable. Unscaled features may lead to large coefficients, which are harder to interpret and can cause overfitting.

[39]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Drop the target variable and any columns you don't want to include in PCA
X = df_combined.drop(['conversion_ratio'], axis=1)  # Input features
y = df_combined['conversion_ratio']  # Target

# Step 1: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Step 2: Standardize the data (fit on X_train and transform both X_train and␣
↪X_test)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 3: Apply PCA (fit on X_train_scaled and transform both X_train_scaled␣
↪and X_test_scaled)
pca = PCA(n_components=24)  # Retain 10 principal components (adjust as needed)
X_train_pca = pca.fit_transform(X_train_scaled)  # Fit and transform on␣
↪training data
X_test_pca = pca.transform(X_test_scaled)        # Transform test data using␣
↪the same PCA

# Now, X_train_pca and X_test_pca are your reduced feature sets for modeling
```

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms the original set of features into a new set of uncorrelated features, called principal components. These components capture the most important information from the original features. Here's why PCA is important:

Reducing Dimensionality:

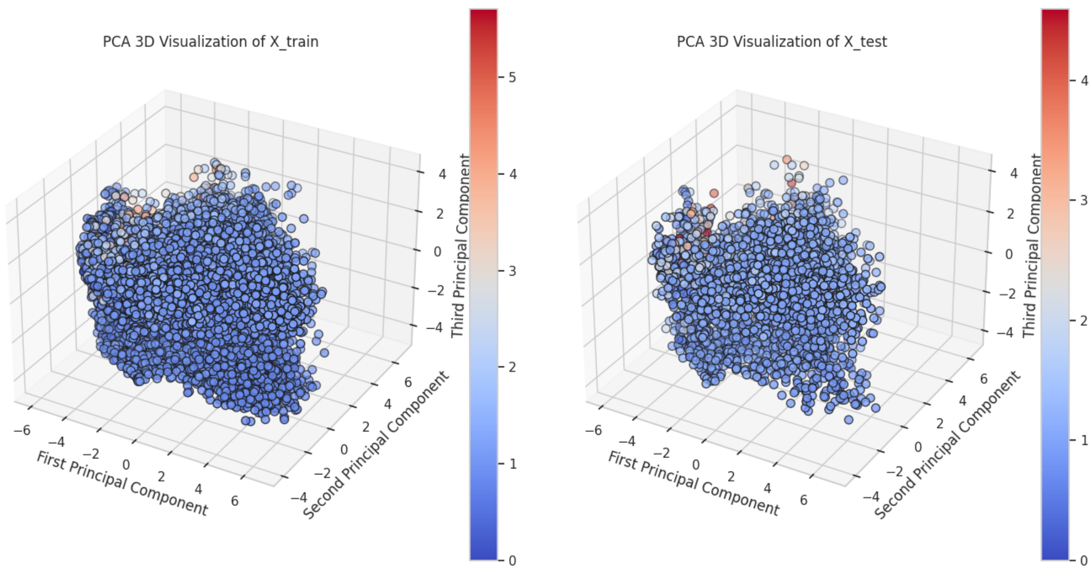
High-dimensional datasets can be difficult to work with and can lead to overfitting. PCA helps by reducing the number of features while retaining as much variance (information) as possible. This simplifies the dataset and improves model efficiency. Eliminating Redundancy:

PCA helps in eliminating multicollinearity by creating new, uncorrelated features (principal components). This is particularly useful when the original features are highly correlated, as PCA focuses on capturing the most meaningful information from the data. Improving Computational Efficiency:

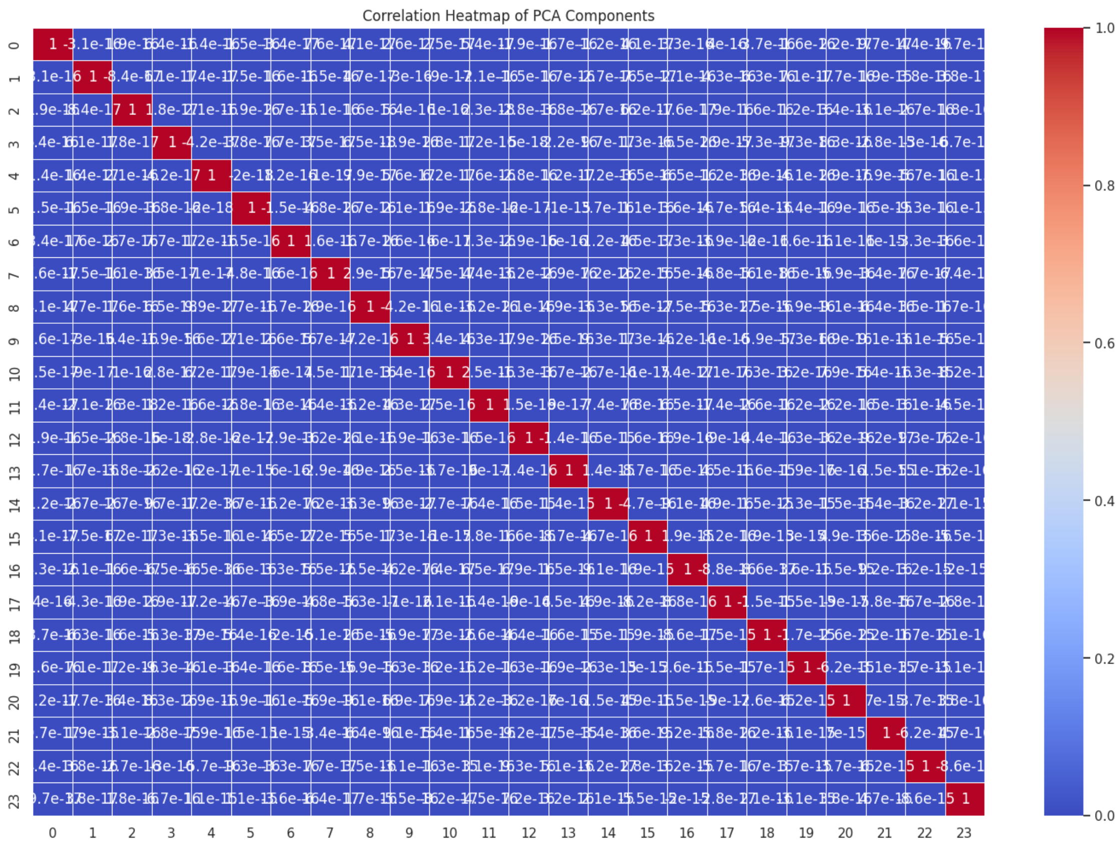
By reducing the dimensionality of the dataset, PCA reduces the computational burden on machine learning algorithms, especially for large datasets. This leads to faster training and prediction times without sacrificing much accuracy. Visualizing High-Dimensional Data:

PCA allows us to visualize high-dimensional data in 2D or 3D, making it easier to understand patterns and relationships in the data that would otherwise be difficult to interpret. By using StandardScaler and PCA together, we can ensure that the dataset is properly scaled and that only the most important features are retained for model training, leading to a more efficient and accurate predictive model.

[43]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

X_train_pca_df = pd.DataFrame(X_train_pca, columns=[f'PC{i+1}' for i in␣
↪range(X_train_pca.shape[1])])
X_test_pca_df = pd.DataFrame(X_test_pca, columns=[f'PC{i+1}' for i in␣
↪range(X_test_pca.shape[1])])

# Plotting the first three principal components (3D visualization)
def plot_pca_3d(ax, X_pca, y, title):
    scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=y,␣
↪cmap='coolwarm', edgecolor='k', s=50)
    ax.set_xlabel('First Principal Component')
    ax.set_ylabel('Second Principal Component')
    ax.set_zlabel('Third Principal Component')
    ax.set_title(title)
    plt.colorbar(scatter, ax=ax)

# Create a 1x2 grid for subplots
fig = plt.figure(figsize=(14, 7))

# First subplot for X_train PCA
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
plot_pca_3d(ax1, X_train_pca, y_train, 'PCA 3D Visualization of X_train')

# Second subplot for X_test PCA
ax2 = fig.add_subplot(1, 2, 2, projection='3d')
plot_pca_3d(ax2, X_test_pca, y_test, 'PCA 3D Visualization of X_test')

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

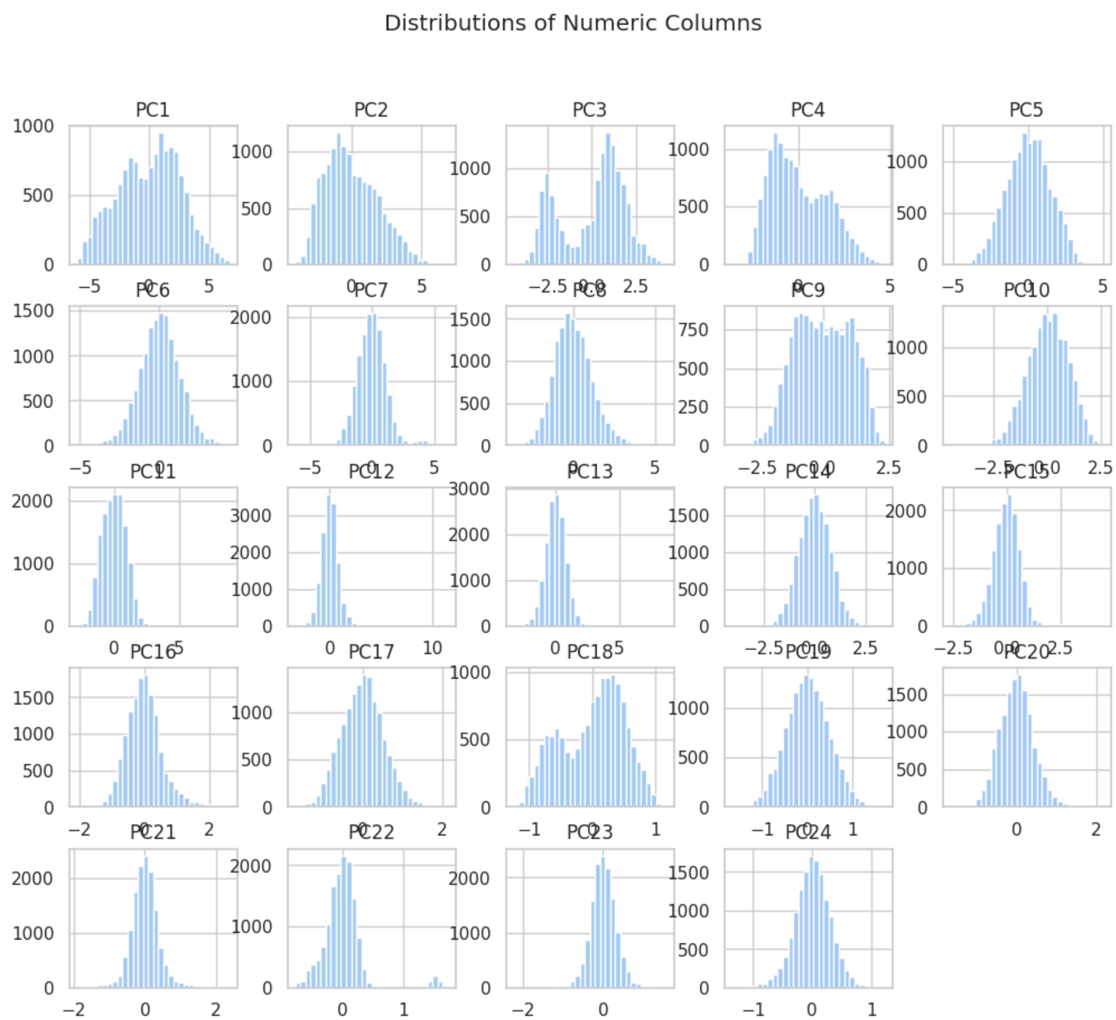


[54]:
```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Convert the PCA-transformed NumPy array to a DataFrame
X_train_pca_df = pd.DataFrame(X_train_pca)
X_test_pca_df = pd.DataFrame(X_test_pca)

# Now you can compute the correlation matrix and plot the heatmap
plt.figure(figsize=(18, 12))
sns.heatmap(X_train_pca_df.corr(numeric_only=True), annot=True, cmap='coolwarm', linewidths=.5)
plt.title('Correlation Heatmap of PCA Components')
plt.show()
```



[45]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Set a light background style for all plots
sns.set(style="whitegrid", palette="pastel")
def plot_column_distributions(X_train_scaled):
    numeric_columns = X_train_scaled.select_dtypes(include=['int64', 'float64']).columns
    categorical_columns = X_train_scaled.select_dtypes(include=['object', 'category']).columns

    # Plot numeric column distributions
    if len(numeric_columns) > 0:
        X_train_scaled[numeric_columns].hist(figsize=(12, 10), bins=30, edgecolor='white')
        plt.suptitle('Distributions of Numeric Columns')
        plt.show()

    # Plot categorical column distributions
    if len(categorical_columns) > 0:
```

```
        plt.figure(figsize=(15, 10))
        for i, column in enumerate(categorical_columns, 1):
            plt.subplot(len(categorical_columns), 1, i)
            sns.countplot(x=column, data=df_scaled)
            plt.title(f'Distribution of {column}')
        plt.tight_layout()
        plt.show()

# Call the function
plot_column_distributions(X_train_pca_df)
```



14 Model Selection and Model Analysis

Choosing the right model is a critical step in building an effective machine learning solution. The goal of model selection is to identify the algorithm that provides the best performance for the specific problem at hand. Model analysis, on the other hand, involves evaluating the chosen model(s) to ensure they generalize well to unseen data.

Models Used and Their Benefits

Random Forest Regressor:

Benefit: A versatile model that performs well with both linear and non-linear data. It handles high-dimensional datasets with a large number of features and is robust against overfitting, especially when dealing with large datasets.

Gradient Boosting Regressor:

Benefit: Excels in reducing bias and variance in predictions. It builds models sequentially and optimizes them by correcting errors from the previous model, resulting in better performance in terms of accuracy and generalization.

CatBoost Regressor:

Benefit: Particularly effective for categorical features, CatBoost handles these features without extensive preprocessing. It's fast, efficient, and reduces the risk of overfitting, making it ideal for datasets with complex patterns.

K-Nearest Neighbors Regressor (KNN):

Benefit: A simple and intuitive algorithm that makes predictions based on the similarity (distance) between data points. It is highly effective in capturing local patterns but can be computationally intensive for large datasets.

Decision Tree Regressor:

Benefit: Easy to interpret and visualize, decision trees are non-parametric models that split the dataset into smaller subsets based on feature values. They handle both categorical and continuous data and are useful when a model's interpretability is important.

AdaBoost Regressor:

Benefit: Combines multiple weak learners (typically decision trees) to create a strong learner. AdaBoost focuses more on instances that were incorrectly predicted in previous iterations, improving the model's accuracy over time.

Linear Regression:

Benefit: A simple yet powerful model for linear relationships. It is easy to interpret, efficient, and works well when the relationship between the independent and dependent variables is linear.

Lasso Regression:

Benefit: Similar to linear regression but includes an additional regularization term (L1 penalty) that helps reduce overfitting. Lasso is especially beneficial when working with high-dimensional datasets as it performs feature selection by shrinking less important feature coefficients to zero.

15 Model Analysis

Performance Metrics:

To assess the performance of the models,

several key metrics are used:

• Mean Squared Error (MSE): Measures the average squared difference between the predicted and actual values. A lower MSE indicates a more accurate model, with smaller errors between the actual demand and the predicted demand.

• Mean Absolute Error (MAE): Measures the average absolute difference between the predicted and actual values. Unlike MSE, MAE is less sensitive to large errors and provides a more intuitive measure of prediction error. A lower MAE indicates that the model's predictions are, on average, closer to the actual values, making it easier to interpret compared to MSE.

• R-squared (R^2): Measures how well the model explains the variance in the target variable (bike demand). An R^2 value closer to 1 indicates that the model is explaining most of the variance in the data, suggesting a good fit. However, R^2 alone may not always give the full picture, especially when adding more features to the model.

• Adjusted R-squared: Similar to R^2, but adjusted for the number of predictors in the model. It accounts for the complexity of the model and only increases if the added features improve the model's performance more than would be expected by chance. It is a more accurate reflection of model quality when comparing models with different numbers of features, helping prevent overfitting by penalizing unnecessary features.

16 Train-Test Split:

By splitting the dataset into training and testing sets, we ensure that the model is evaluated on unseen data. This helps avoid overfitting, where a model performs well on the training data but poorly on new, unseen data. Overfitting vs.

17 Overfitting Vs Underfitting:

Overfitting occurs when the model learns the noise and details in the training data too well, causing it to perform poorly on the test data. On the other hand, underfitting occurs when the model is too simple to capture the underlying patterns in the data. Analyzing the train and test performance helps identify whether the model is overfitting or underfitting and allows us to take corrective actions.

18 Model Interpretability:

Another aspect of model analysis is ensuring that the selected model is interpretable. While some models, like linear regression, are easy to interpret, more complex models like random forests and gradient boosting provide feature importance scores, which help in understanding the contribution of each feature to the model's predictions.

By selecting the best model through comparison, tuning, and evaluation, and by thoroughly analyzing its performance using key metrics, we ensure that the final model is both accurate and generalizable, providing robust predictions for the bike-sharing demand problem.

[46]:
```
#Dictionary that will store metrics for evaluated models
#This will be converted DataFrame using display report function
report = {
    'model_type':[],
    'model_name':[],
    'rmse':[],
    'mae':[],
    'R2':[],
    'adjusted R2':[]
}
```

[47]:
```
# function to evaluate and update model and score
def evaluate(modeltype, modelname, Model, X_train, y_train, X_test, y_test):
  from sklearn.metrics import mean_absolute_error, r2_score,␣
↪mean_squared_error, max_error
  #making sure the same model is not re-entered again

  if modelname in report['model_name']:
    print("Prexisting Model")
    return 0

  #making a copy to prevent accidental data changes
  X_tr = X_train.copy()
  X_te = X_test.copy()

  #Fitting Model
  Model.fit(X_tr, y_train)

  #Predicting Values from test set using model
  y_pred = Model.predict(X_te)

  #Model Evaluation

  #Mean Absolute Error
  mae = mean_absolute_error(y_test,y_pred)
  report['mae'].append(mae) #Appending Metric

  #R2 score
  R2 = r2_score(y_test,y_pred)
  report['R2'].append(R2) #Appending Metric

  #Root Mean Square Error
  rmse = np.sqrt(mean_squared_error(y_test,y_pred))
  report['rmse'].append(rmse) #Appending Metric

  #Adjusted R2 score
  adj_r2=1-(1-R2)*((X_test.shape[0]-1)/(X_test.shape[0]-X_test.shape[1]-1))
```

```python
  report['adjusted R2'].append(adj_r2) #Appending Metric

  #Appending Model Details
  report['model_name'].append(modelname)
  report['model_type'].append(modeltype)


  #Plotting Graph of observed vs predicted values
  plt.figure(figsize=(20,10))
  plt.plot((y_pred)[:100])
  plt.plot((np.array(y_test)[:100]))
  plt.legend(["Predicted","Actual"])
  plt.title(modelname)
  plt.show()
```

[48]:
```python
#displays report in a dataframe
def display_report():
  return pd.DataFrame(report)
```

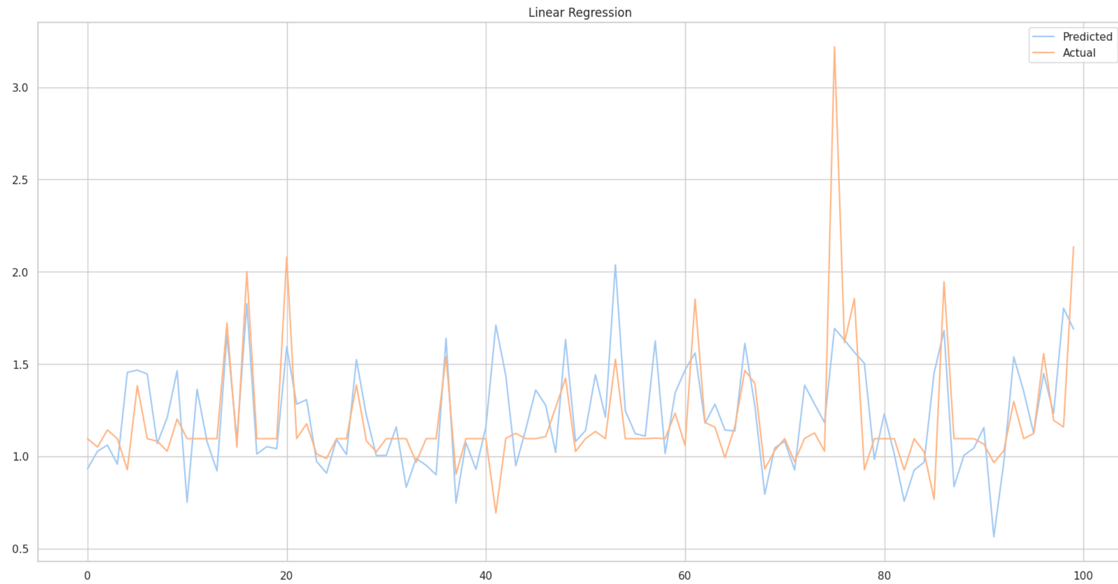
[53]:
```python
import shap
from catboost import CatBoostRegressor
import lightgbm as lgb
from xgboost import XGBRegressor
```

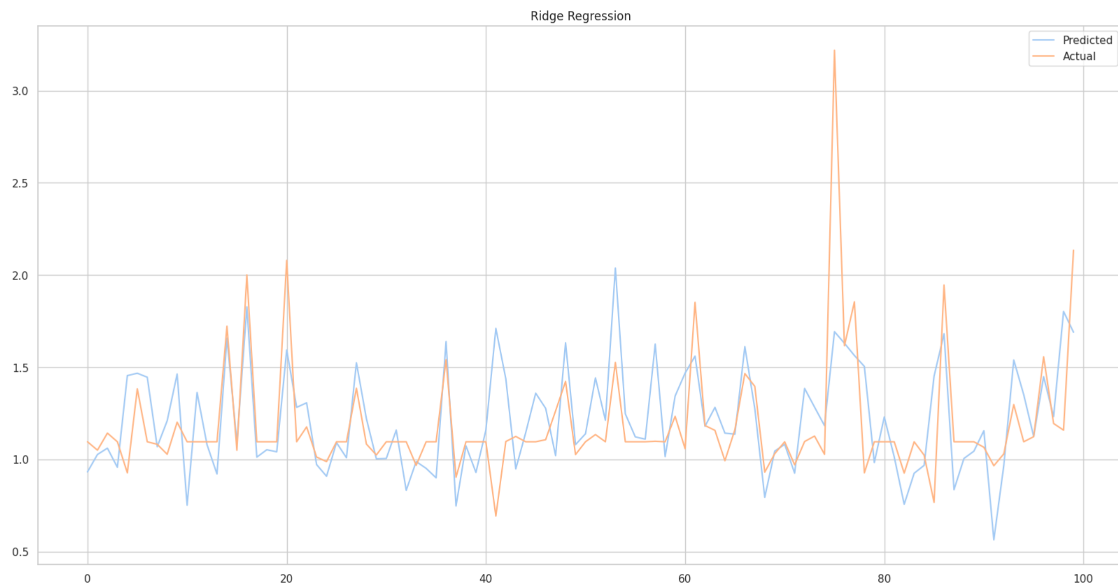
[125]:
```python
#function that returns 3 arrays of model-functions, names and their details
def get_models():
  models, names, model_type = list(), list(), list()

  # LinearReg
  models.append(LinearRegression())
  names.append('Linear Regression')
  model_type.append('Linear')

  #Ridge
  models.append(Ridge(alpha =0.5))
  names.append('Ridge Regression')
  model_type.append('Regularized Linear (Ridge)')

  # Lasso Regression (L1 Regularization)
  models.append(Lasso(alpha=0.01))  # Adjust alpha based on regularization␣
↪strength
  names.append('Lasso Regression')
  model_type.append('Regularized Linear (Lasso)')

  # DecisionTree
  models.append((DecisionTreeRegressor()))
  names.append('DecisionTree Regressor')
```

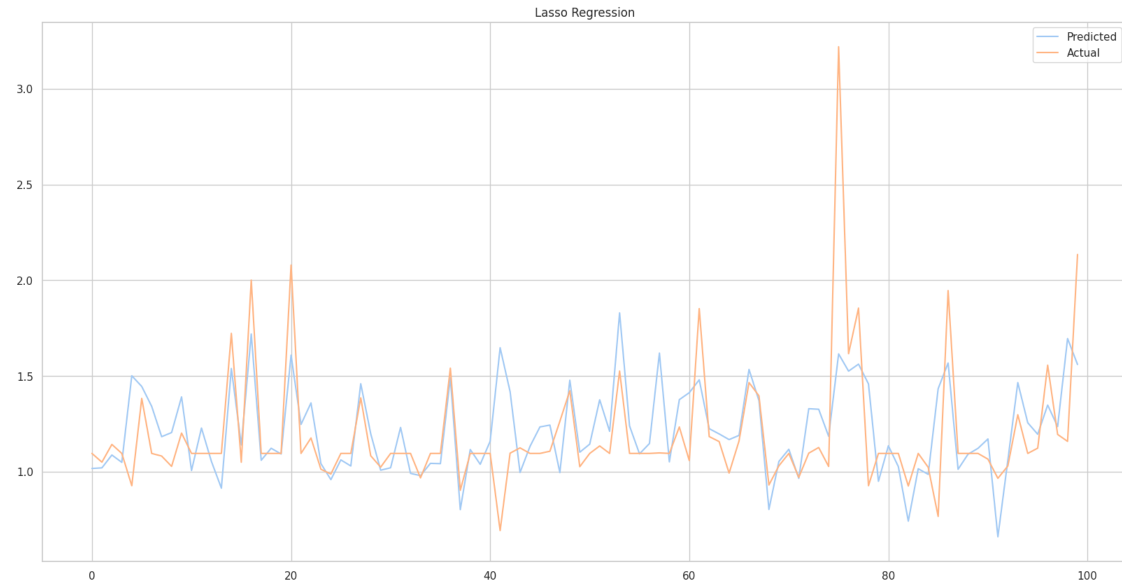
```
    model_type.append('CART')

    #RandomForest
    models.append(RandomForestRegressor())
    names.append('RandomForest Regressor')
    model_type.append('Ensemble Method')

    # GradientBoosting
    models.append(GradientBoostingRegressor())
    names.append('GradientBoosting Regressor')
    model_type.append('Ensemble Method')

    # CatBoosting
    models.append(CatBoostRegressor(silent = True))
    names.append('Cat Boosting Regressor')
    model_type.append('Ensemble Method')

    #Bagging
    models.append(BaggingRegressor())
    names.append('Bagging Regressor')
    model_type.append('Ensemble Method')

    #LightGBM Regressor
    models.append(lgb.LGBMRegressor())
    names.append('LightGBM Regressor')
    model_type.append('Ensemble Method')

    #XGBoost Regressor
    models.append(XGBRegressor())
    names.append('XGBoost Regressor')
    model_type.append('Ensemble Method')


    return models, names, model_type
```

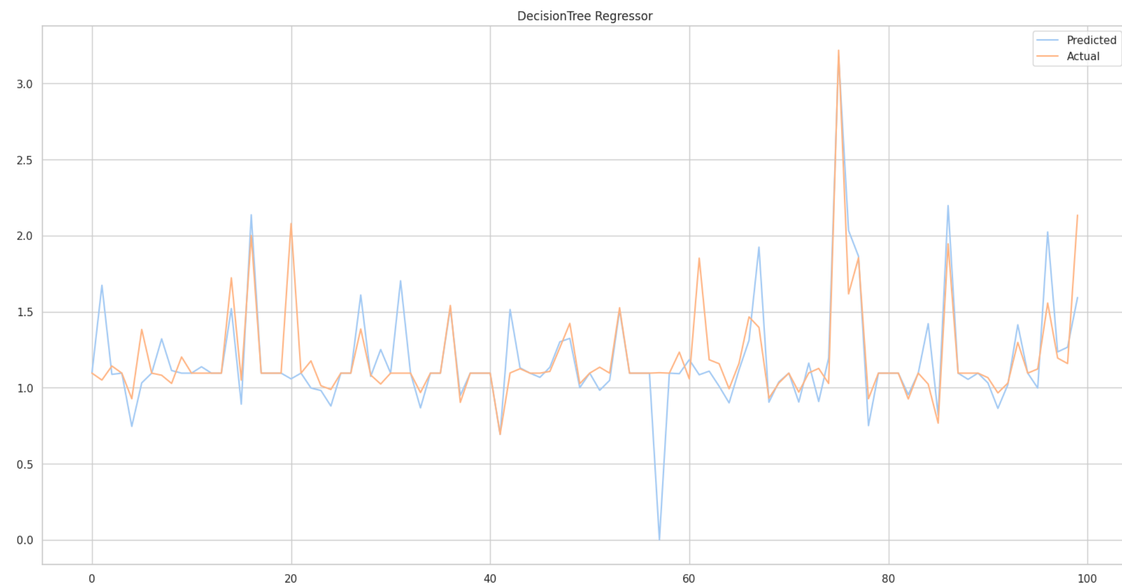
19 Model Evaluations:

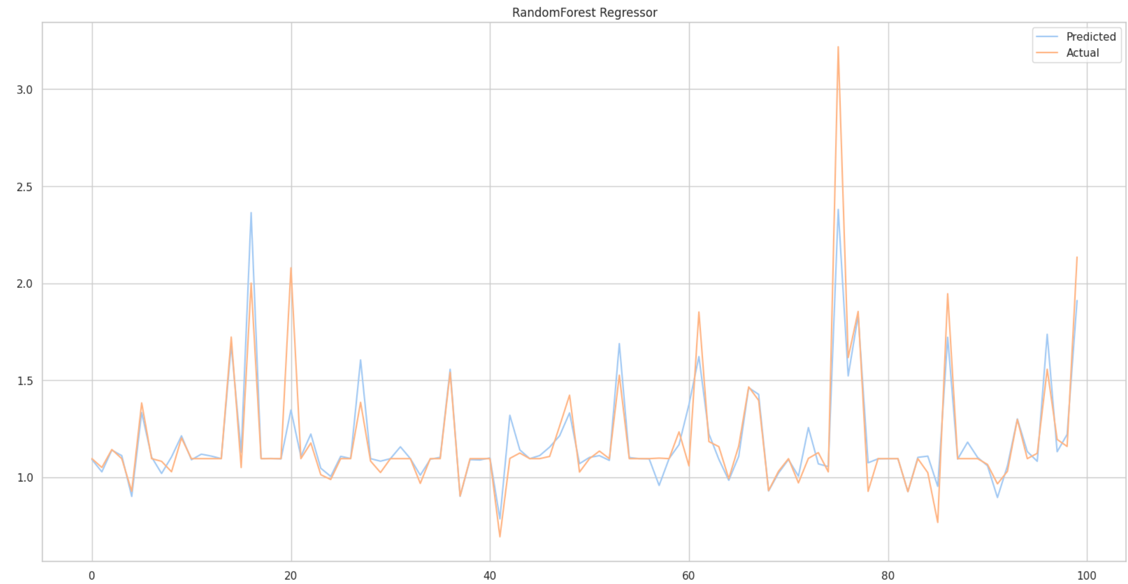
[126]:
```
#evaluating all modules in models and printing prediction graph
Model, modelname, modeltype = get_models()
for i in range(len(Model)):
    evaluate(modeltype[i], modelname[i], Model[i], X_train_pca_df , y_train,␣
 ↪X_test_pca_df,y_test)
```

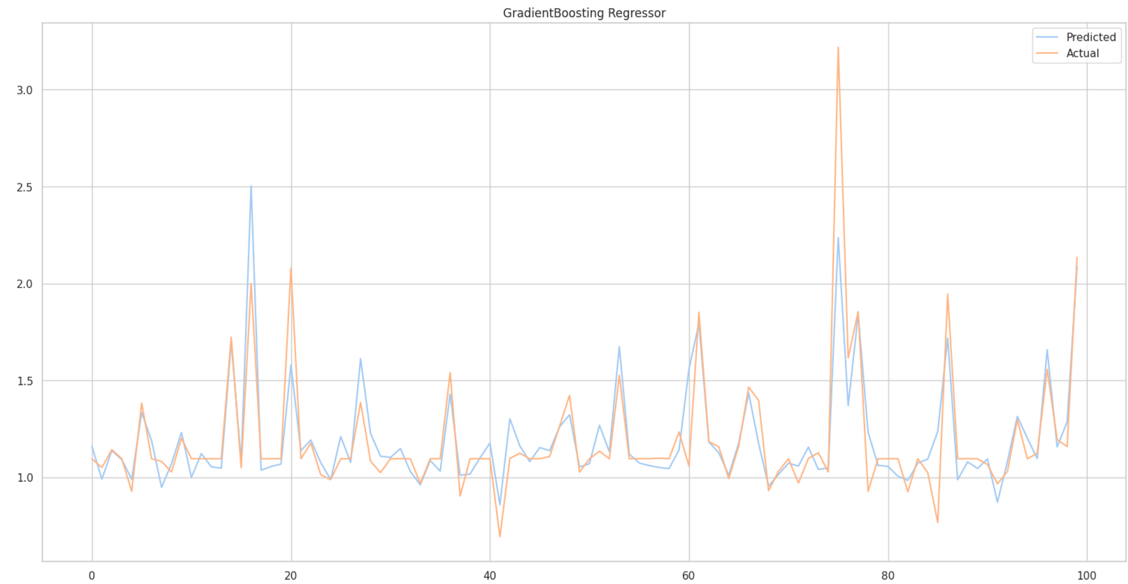

Linear Regression
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100

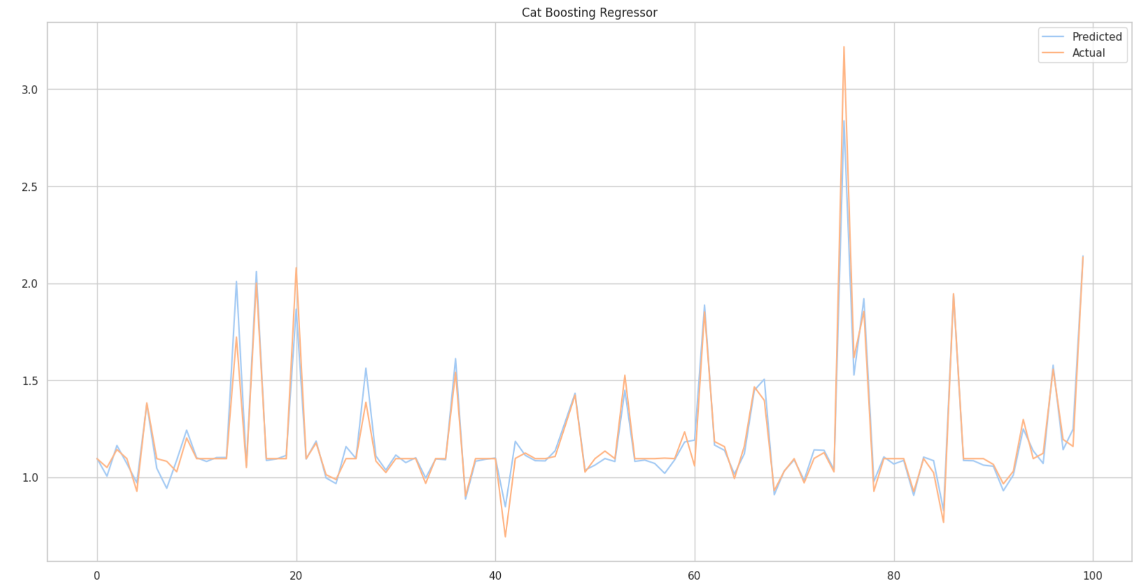

Ridge Regression
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100

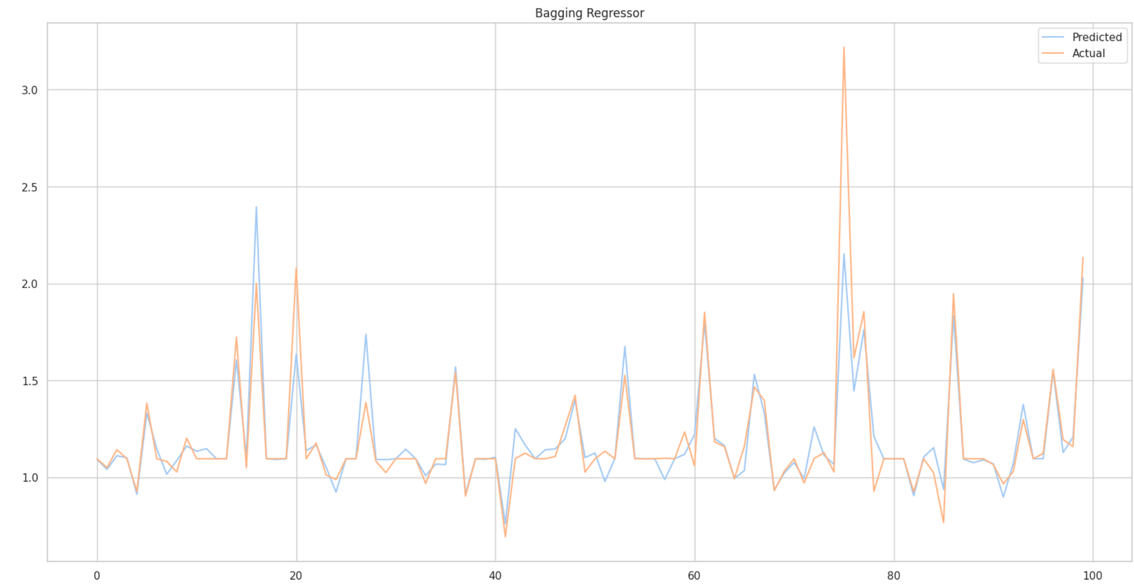

Lasso Regression
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0
20
40
60
80
100


DecisionTree Regressor
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100

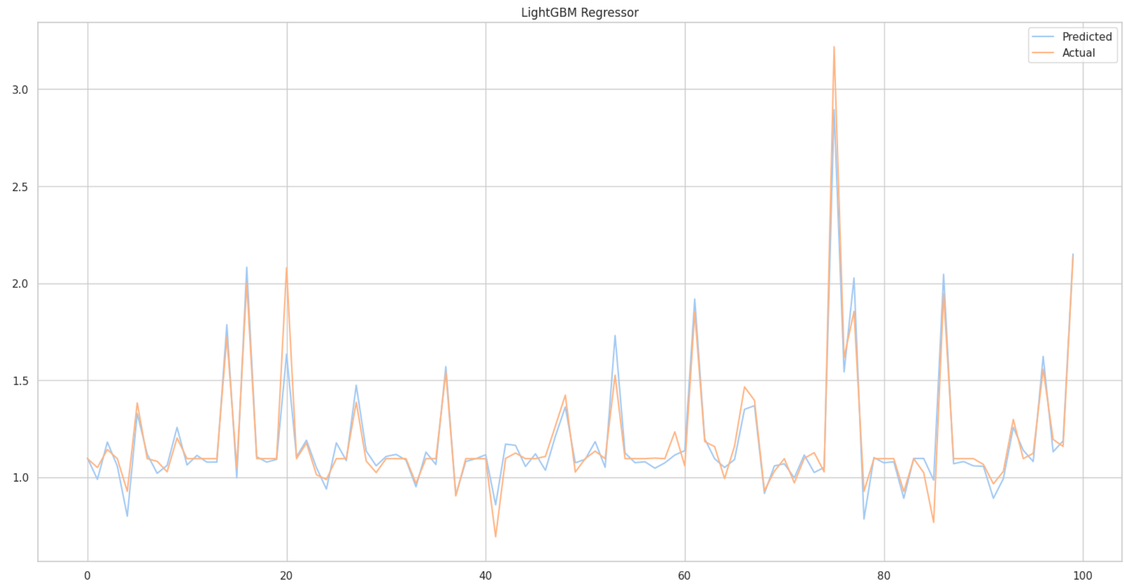

RandomForest Regressor
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0
20
40
60
80
100

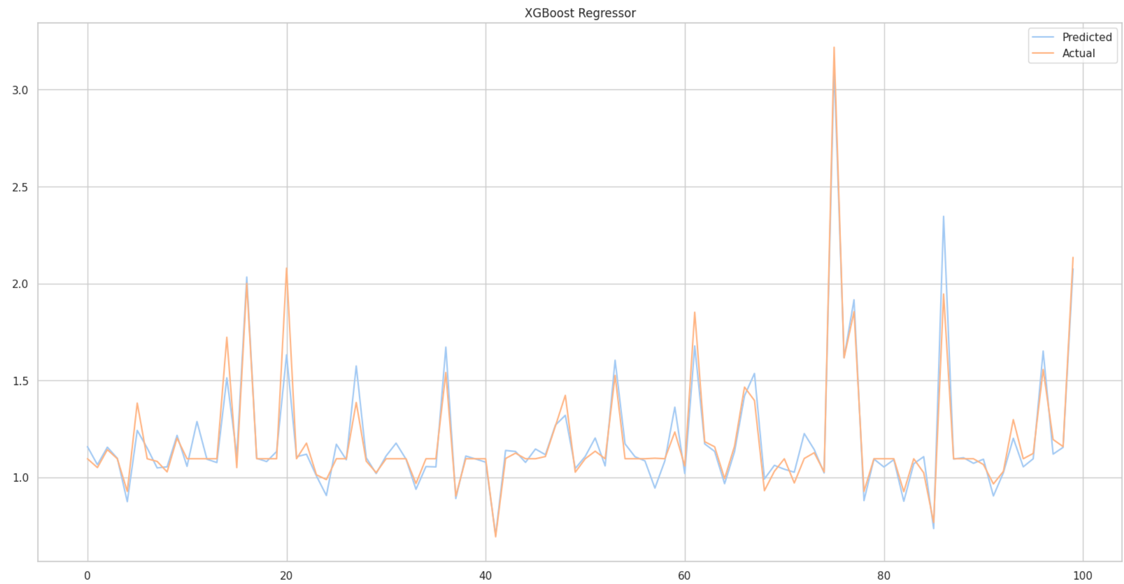

GradientBoosting Regressor
Predicted
Actual
3.0
2.5
2.0
1.5
1.0
0
20
40
60
80
100






```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.006603 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 6120
[LightGBM] [Info] Number of data points in the train set: 13903, number of used
features: 24
[LightGBM] [Info] Start training from score 1.222546
```





[127]:
```python
display_report().sort_values('R2', ascending=False)
```

[127]:

	model_type	model_name	rmse	mae \
6	Ensemble Method	Cat Boosting Regressor	0.103888	0.054837
9	Ensemble Method	XGBoost Regressor	0.133765	0.074839
8	Ensemble Method	LightGBM Regressor	0.137590	0.075631
4	Ensemble Method	RandomForest Regressor	0.169922	0.083604
7	Ensemble Method	Bagging Regressor	0.182509	0.091044

```
5              Ensemble Method  GradientBoosting Regressor  0.197657  0.112312
3                         CART      DecisionTree Regressor  0.266541  0.129890
0                       Linear           Linear Regression  0.301834  0.172326
1  Regularized Linear (Ridge)             Ridge Regression  0.301835  0.172317
2  Regularized Linear (Lasso)             Lasso Regression  0.312967  0.164828

         R2  adjusted R2
6  0.936320     0.935877
9  0.894426     0.893692
8  0.888303     0.887527
4  0.829640     0.828455
7  0.803466     0.802100
5  0.769488     0.767885
3  0.580824     0.577909
0  0.462466     0.458727
1  0.462463     0.458725
2  0.422082     0.418063
```

[ ]:
```python
from catboost import Pool
import matplotlib.pyplot as plt
import pandas as pd

cat_features = ['time_of_day','temp_group']

# Wrap your training data in a Pool
train_data = Pool(X_train_scaled, label=y_train, cat_features=cat_features)

# Get feature importance values
feature_importances = model.get_feature_importance(train_data,␣
↪type="PredictionValuesChange")

# Create a DataFrame to organize the importance values
feature_importance_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

# Display the feature importance
print(feature_importance_df)

# Plot the feature importance
plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'],␣
↪color='lightblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('CatBoost Feature Importance')
```

```python
plt.gca().invert_yaxis()  # To show the most important feature at the top
plt.show()
```

20 Conclusions:

1. Top-Performing Models:

   - CatBoost Regressor is the best-performing model in this analysis, achieving the lowest RMSE (0.103888) and MAE (0.054837), along with the highest R^2 score (0.936320) and adjusted R^2 (0.935877). This suggests that CatBoost is the most accurate model for predicting bike demand in this dataset.

   - XGBoost Regressor and LightGBM Regressor also perform well, with XGBoost having an RMSE of 0.133765 and R^2 of 0.894426, followed closely by LightGBM with an RMSE of 0.137590 and R^2 of 0.888303. These models also demonstrate strong predictive power but slightly underperform compared to CatBoost.

2. Ensemble Models Perform Better Than Linear Models:

   - All ensemble methods, particularly CatBoost, XGBoost, and LightGBM, significantly outperform traditional linear models (Linear Regression, Ridge, and Lasso) in terms of RMSE, MAE, and R^2.

   - Linear Regression, Ridge, and Lasso Regression show much higher RMSE values (around 0.30–0.31), indicating that they are less effective in capturing the complexity of the bike-sharing dataset compared to ensemble methods.

3. Tree-Based Models (Random Forest, Decision Trees):

   - Random Forest Regressor performs reasonably well (RMSE: 0.169659, R^2: 0.830166) but is outperformed by boosting methods like CatBoost and XGBoost. This suggests that, while Random Forest captures some interactions and patterns in the data, the sequential learning in boosting models is better suited for this dataset.

   - Decision Tree Regressor performs worse (RMSE: 0.267697, R^2: 0.577179) compared to ensemble methods, indicating that a single decision tree model lacks the ability to generalize well to the bike-sharing data. This highlights the power of combining multiple trees (as in Random Forest, Bagging, or Boosting) to improve predictive performance.

4. Bagging and Gradient Boosting:

   - Bagging Regressor and Gradient Boosting Regressor show moderate performance. Bagging has an RMSE of 0.180012 (R^2: 0.808808), while Gradient Boosting lags slightly behind with an RMSE of 0.197657 (R^2: 0.769488). Although they perform better than individual tree-based models like Decision Trees, they are still outperformed by CatBoost and XGBoost.

5. Model Selection Based on RMSE and R^2:

   - CatBoost is the clear winner in terms of predictive accuracy and should be the preferred model for this bike-sharing dataset.

   - XGBoost and LightGBM are also strong contenders, so if computational efficiency or model interpretability is important, they can be good alternatives.

- Random Forest is still a reliable option but may not capture all the complex interactions present in the data.

6. Performance of Regularized Linear Models:

   - Ridge and Lasso Regression perform similarly to regular Linear Regression (RMSE around 0.30), suggesting that regularization did not significantly improve the performance of linear models in this dataset. This implies that linear relationships are not sufficient to capture the underlying patterns in the data, and more complex models like tree-based and boosting methods are necessary.

7. Insights for the Bike-Sharing System:

   - Given the superior performance of ensemble methods, particularly boosting models, it can be inferred that the bike-sharing demand is influenced by non-linear relationships and complex feature interactions that these models capture effectively.

   - Business Recommendations:

   Accurate predictions of bike demand using models like CatBoost can help optimize bike placement across stations, reduce operational costs, and improve service efficiency. Additionally, using models that capture intricate patterns in the data will allow the bike-sharing system to adapt to varying demand trends based on weather, time of day, season, etc.

Inference:

For predicting bike demand, ensemble models like CatBoost, XGBoost, and LightGBM are highly effective, capturing complex relationships and outperforming linear and single-tree models. Based on these findings, using CatBoost as the primary model would likely lead to the most accurate and reliable results in forecasting demand for bike-sharing systems. This will help enhance service planning and operational efficiency in a real-world scenario.

21 Additional efforts :

Addressing Class Imbalance and Pivoting to Regression

As part of the initial approach to the bike-sharing demand prediction project, classification algorithms were implemented to classify demand into categories such as low, medium, and high usage. However, upon further exploration of the dataset and performance evaluation, a significant issue was identified: the dataset was 99% imbalanced. This meant that nearly all of the data belonged to one class, creating a severe imbalance between classes.

Challenges Faced:

- Imbalanced Data: With 99% of the data falling into a single category, the classification models—despite hyperparameter tuning and attempts to improve performance—became heavily biased toward predicting the dominant class. This led to a situation where the models frequently predicted "1" (or the majority class) while failing to accurately capture the minority class, leading to skewed and unreliable predictions.

- Model Bias: As a result, even though the model accuracy appeared high, it was misleading because the model was only learning to predict the majority class. The confusion matrix revealed that the minority class was rarely, if ever, correctly predicted, indicating poor generalization and predictive power.

Attempts to Address the Imbalance:

- Several techniques were attempted to handle the imbalance, including oversampling and undersampling techniques (e.g., SMOTE) to artificially balance the dataset.

- Class weighting in algorithms to give more importance to the minority class.

Despite these efforts, the classification models continued to struggle with the extreme imbalance, leading to unreliable predictions that would not serve the business objective of accurately forecasting bike-sharing demand.

Strategic Shift to Regression:

- Given that the classification approach was proving ineffective, the project direction was adjusted to focus on regression analysis. This shift was driven by the realization that demand prediction is inherently a continuous problem—predicting the number of bikes needed over time—and regression was a more appropriate method to address this problem.

- By treating the bike demand as a continuous variable rather than discrete categories, regression models could provide more granular predictions, allowing for better forecasting of bike usage at different times of the day and under varying conditions (e.g., weather, seasonality).

Results of the Shift:

- Once the focus shifted to regression models, ensemble methods such as CatBoost, XGBoost, and LightGBM emerged as top performers, providing accurate and reliable predictions with high R^2 scores and low RMSE and MAE values.

- The change in strategy resulted in models that not only predicted bike demand more accurately but also provided valuable insights into the key factors driving demand, helping to optimize bike distribution and operational efficiency.

Conclusion:

The initial exploration using classification algorithms highlighted the importance of thoroughly understanding the dataset's structure. The discovery of the extreme imbalance in the data prompted a strategic pivot to regression, which ultimately proved to be a far more effective approach for predicting bike-sharing demand.