Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Intelligent Systems Laboratory
ENCS5141

Case Study #1

Data Cleaning and Feature Engineering for the Bike Sharing Dataset

Prepared by

Usama Shoora 1200796

Instructor

Dr. Mohammad Jubran

Teacher assistant

Eng. Hanan Awawdeh

Section **1**

BIRZEIT
August – 2024

Abstract

This report delves into the process of data cleaning and feature engineering using the Bike Sharing Dataset. It encompasses various methodologies and techniques used to preprocess the data, including handling missing values, outlier detection, and encoding categorical variables. Through visualizations, the report illustrates the impact of different features on bike rental counts, particularly focusing on weather conditions and temporal factors. Feature selection and dimensionality reduction techniques, such as PCA, are applied to optimize the dataset for model training. The ultimate goal is to enhance the understanding of the data and improve the performance of predictive models in the context of bike sharing.

Table of Contents

List of Figures

List of Listings

List of Tables

1. Introduction

The study of bike sharing systems has gained significant traction due to their role in promoting sustainable urban transportation. This report focuses on the Bike Sharing Dataset, aiming to prepare the data for predictive modelling through a series of data cleaning and feature engineering tasks.

The initial steps involve loading and exploring the dataset to understand its structure and the nature of its features. This includes presenting a summary of the data, examining its basic statistical properties, and identifying any initial patterns or anomalies. Understanding the dataset at this level is crucial for informing subsequent data processing steps.

Subsequent sections detail the processes of handling missing data and outliers, ensuring that the dataset is as complete and accurate as possible. Handling missing data may involve techniques such as imputation, where missing values are filled in using statistical methods or derived from other available data. Outliers, which can skew analysis and modelling, are identified and addressed to maintain data integrity.

The transformation of categorical features into numerical formats suitable for analysis is another key task. This often involves encoding techniques such as one-hot encoding, where categorical variables are converted into a format that can be readily used in machine learning algorithms. This step is vital for ensuring that all features in the dataset can be effectively analyzed and used in predictive models.

Visualization techniques are employed throughout to uncover patterns and relationships within the data. By creating visual representations of the data, such as plots and charts, we can gain insights into user behavior and environmental influences on bike rentals. These visualizations help to illustrate how factors like season, weather, and time of day impact bike rental usage.

The report culminates in the application of feature selection and dimensionality reduction techniques to streamline the dataset for model training. Techniques such as Principal Component Analysis (PCA), and SelectKBest are used to reduce the dataset to its most relevant features, minimizing noise and improving computational efficiency. This process enhances the predictive accuracy of models trained on the dataset by focusing on the most significant factors influencing bike sharing usage.

2. Procedure and Discussion

2.1. Importing the Necessary Dependencies

```
import pandas as pd
import numpy as np
from numpy import isnan

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import LabelEncoder, OneHotEncoder, MinMaxScaler
from sklearn.impute import KNNImputer
from sklearn.model selection import train test split, cross val score
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import SelectKBest, mutual_info_regression
from sklearn.metrics import r2_score
from sklearn.decomposition import PCA
```

Listing 2.1: Importing Necessary Dependencies

The dependencies (libraries) listed above are crucial for performing the study.

2.2. Loading the Bike-Sharing Dataset

```
!git clone https://github.com/mkjubran/ENCS5141Datasets.git

data = pd.read_csv("/content/ENCS5141Datasets/ENCS5141_BikeSharingDataset_Modified/hours.csv")
data = pd.DataFrame(data)
data.head()
```

Listing 2.2: Loading the Dataset

The dataset is loaded into the data frame ‘data’. The first five records of *data* are shown below.

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	1/1/11	Spring	0	1.0	0	0.0	Saturday	0	1.0	0.24	0.2879	0.81	0.0	3.0	13.0	16.0
1	2	1/1/11	Spring	0	1.0	1	0.0	Saturday	0	1.0	0.22	0.2727	0.80	0.0	8.0	32.0	40.0
2	3	1/1/11	Spring	0	1.0	2	0.0	Saturday	0	1.0	0.22	0.2727	0.80	0.0	5.0	27.0	32.0
3	4	1/1/11	Spring	0	1.0	3	0.0	Saturday	0	1.0	0.24	0.2879	0.75	0.0	3.0	10.0	13.0
4	5	1/1/11	Spring	0	1.0	4	0.0	Saturday	0	1.0	0.24	0.2879	0.75	0.0	0.0	1.0	1.0

Table 2.1: Original Data Head

Right away, we can see that the dataset has 17 features. Including numerical, categorical, and non-numeric features.

- **Instant:** Index of record.
- **Dteday:** Date of record.
- **Season:** Season record is in.
- **Yr:** Year of record.
- **Mnth:** Month of record.
- **Hr:** Hour of record.
- **Holiday:** If record is during a holiday.
- **Weekday:** Day of record.
- **Workingday:** If record on a workday.
- **Weathersit:** Weather condition
- **Temp:** Temperature
- **Atemp:** Apparent temperature.
- **Hum:** Humidity.
- **Windspeed:** Speed of wind.
- **Casual:** # unregistered users.
- **Registered:** # registered users.
- **Cnt:** casual and registered sum

2.3. Data Exploration

To understand the dataset better, the following code listings are executed.

```
data.shape
```

Listing 2.3: Displaying Data Shape

The shape of the data is (17379, 17). Meaning it had 17379 records, and 17 columns.

```
data.info()
```

Listing 2.4: Displaying Data Info

The output of the code is as follows:

```
RangeIndex: 17379 entries, 0 to 17378
Data columns (total 17 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instant     17379 non-null  int64
 1   dteday      17379 non-null  object
 2   season      17379 non-null  object
 3   yr          17379 non-null  int64
 4   mnth        17378 non-null  float64
 5   hr          17379 non-null  int64
 6   holiday     17367 non-null  float64
 7   weekday     17378 non-null  object
 8   workingday  17379 non-null  int64
 9   weathersit  17376 non-null  float64
 10  temp        16930 non-null  float64
 11  atemp       17324 non-null  float64
 12  hum         17086 non-null  float64
 13  windspeed   17071 non-null  float64
 14  casual      17344 non-null  float64
 15  registered  17363 non-null  float64
 16  cnt         17367 non-null  float64
```

Listing 2.5: Data Info

Most of the columns have numeric data, while others are non-numeric, and some are categorical. Some features also have missing data, as their Non-Null Count is less than 17379.

Some features are hard to read, and others aren't really indicative/useful for the purpose of this case study. Renaming and dropping unnecessary features is done by executing the code listings below.

```
data = data.rename(columns={'weathersit':'weather',
                            'yr':'year',
                            'mnth':'month',
                            'hr':'hour',
                            'hum':'humidity',
                            'cnt':'count'})

data.drop(columns = ['instant' , 'dteday' , 'year'], inplace=True)
```

Listing 2.6: Rename Features and Delete Unnecessary Features Code

Now the features are more readable and easier to understand, and data such as instant (which is the index of records), dteday (date of the record), and year (year of the record) have been dropped since they aren't useful.

The categorical features should be defined accordingly, which can be done by executing the following code.

```
cols = ['season' , 'month' , 'hour' , 'holiday' , 'weekday' , 'workingday' , 'weather']

for col in cols:
    data[col] = data[col].astype('category')

data.info()
```

Listing 2.8: Defining Categorical Features

```
RangeIndex: 17379 entries, 0 to 17378
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   season      17379 non-null  category
 1   month       17378 non-null  category
 2   hour        17379 non-null  category
 3   holiday     17367 non-null  category
 4   weekday     17378 non-null  category
 5   workingday  17379 non-null  category
 6   weather     17376 non-null  category
 7   temp         16930 non-null  float64
 8   atemp        17324 non-null  float64
 9   humidity    17086 non-null  float64
 10  windspeed   17071 non-null  float64
 11  casual       17344 non-null  float64
 12  registered  17363 non-null  float64
 13  count        17367 non-null  float64
```

Listing 2.7: Necessary Data Info

Now the data contains only the necessary data with better names, and accurate types.

To learn more about the missing data, the following code is executed.

```
missing_values = data.isnull().sum()
missing_values[missing_values > 0]
missing_values[missing_values > 0].to_frame().T
```

Listing 2.9: Number of Missing Data Code

The output is as follows:

Table 2.2 Number of Missing Data

Feature	month	holiday	weekday	weather	temp	atemp	humidity	windspeed	casual	registered	count
# Missing Values	1	12	1	3	449	55	293	308	35	16	12

We can see that these 11 features have missing data in varying amounts, they will be handled in upcoming sections.

2.4. Data Visualization

2.4.1. Categorical Features Plots

Bar plots can be plotted for the categorical features as follows.

```
plt.figure(figsize=(5,5))
counter=1
plt.figure(figsize=(15,15))

cols = ['season' , 'month' , 'hour' , 'holiday' , 'weekday' , 'workingday' , 'weather']

for column in cols:
  plt.subplot(3,3,counter)
  sns.barplot(x=column, data=data, y='count', hue=column, legend=False, estimator=sum)
  plt.title(f"Count vs {column.capitalize()}")
  counter+=1
plt.show()
```

Listing 2.10: Barplots of Categorical Features Code

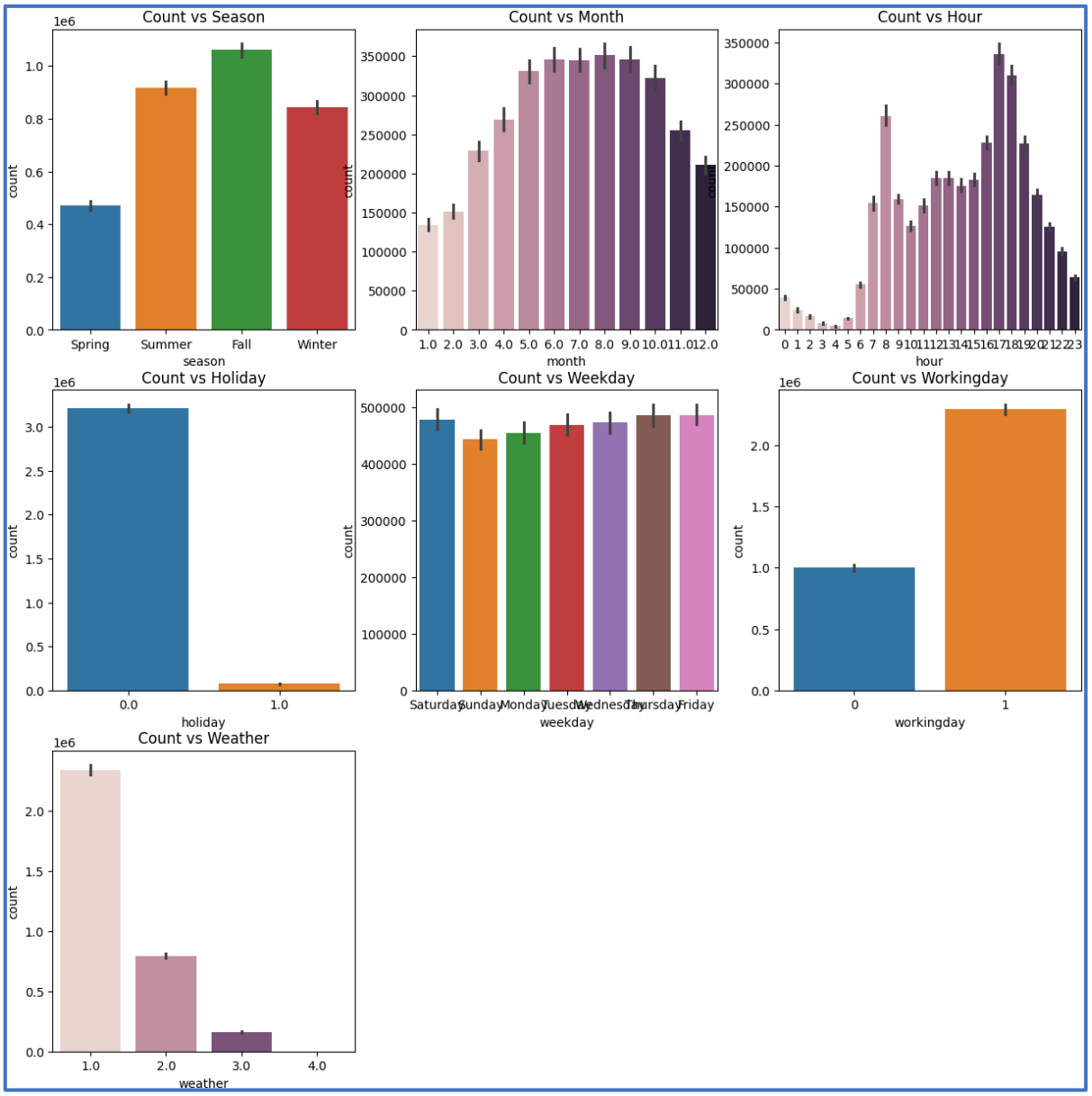

Figure 2.1: Barplots of Categorical Features

From the previous plots, some interesting observations can be noted.

- **Count vs Season:** Bike rentals peak during Fall and are lowest during Spring. Which means Fall is the peak season, possibly for its weather conditions that suit biking most.
- **Count vs Month:** Bike rentals increase steadily from January till June, then stay at the peak around mid-year, then decline towards December. Indicating that warmer months tend to have more bike rentals.
- **Count vs Hour:** Bike rentals peak in the morning around 7-9 AM, and in the evening around 5-6 PM, then drop during the night. Which corresponds to typical peak and commuting hours in general.
- **Count vs Holiday:** Bike rentals are much higher on non-holidays than on holidays. Suggesting that bike rentals are more likely used for commuting and errands, rather than for fun and weekends.
- **Count vs Weekday:** Bike rentals are considerably high on all weekdays with slight variations between them. Which might suggest a steady demand for bike sharing during the workday, likely because of regular commuters/users. The slight variations may result from different work schedules or from the weather on particular days.
- **Count vs Workingday:** Bike rentals are higher, more than double, on working days compared to non-working days. This indicates that bike sharing is mostly utilized for commuting, which is consistent with the pattern shown in the "**Count vs Holiday**" plot.
- **Count vs Weather:** Bike rentals are highest in clear or fair-weather conditions, and decline as weather conditions worsen. This shows how weather plays a big role in the number of users.

2.4.2. User Type Plot

The relation between registered and unregistered (casual) users against the total count can be plotted as follows.

```
sns.scatterplot(data=data, x='registered', y='count', label='Registered', marker='o')
sns.scatterplot(data=data, x='casual', y='count', label='Casual', marker='o')

plt.xlabel('Users')
plt.ylabel('Count')
plt.title('Registered and Casual Users vs Count')
plt.legend(title='User Type')
plt.show()
```

Listing 2.11: User Type Plot Code

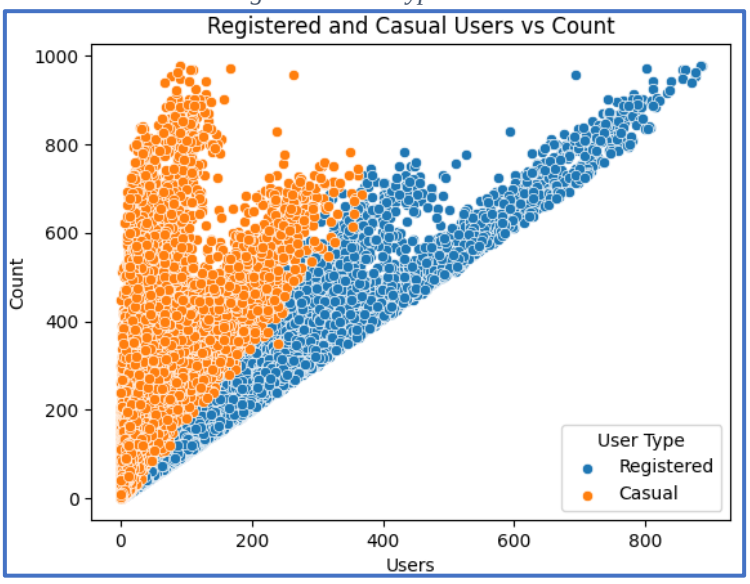

Figure 2.2: User Type vs Count Plot

Registered users have a greater impact on the total count because their rental numbers are generally higher. Casual users, on the other hand, have points clustered in lower counts, demonstrating a lower correlation between casual users and total count.

2.4.3. Counts vs Hour Plots (Grouped by Several Features)

```
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

sns.barplot(data=data, x='hour', y='count', hue='hour', ax=axes[0], legend=False)
axes[0].set_xlabel('Hour')
axes[0].set_ylabel('Count')
axes[0].set_title('Count vs Hour')

sns.pointplot(data=data, x='hour', y='count', hue='weekday', ax=axes[1])
axes[1].set_xlabel('Hour')
axes[1].set_ylabel('Count')
axes[1].set_title('Count vs Hour')

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

Listing 2.12: Counts vs Hour Plots (Grouped by weekdays) Code

- To plot the total count of bike rentals per hour according to the weekdays, the following code is run.

Bike rentals peak in the morning around 7-9 AM, and in the evening around 5-6 PM, then drop during the night. Which corresponds to typical peak and commuting hours in general. But there's a bump during mid-noon on the weekends. Plotting casual and registered users separately could help explain it.

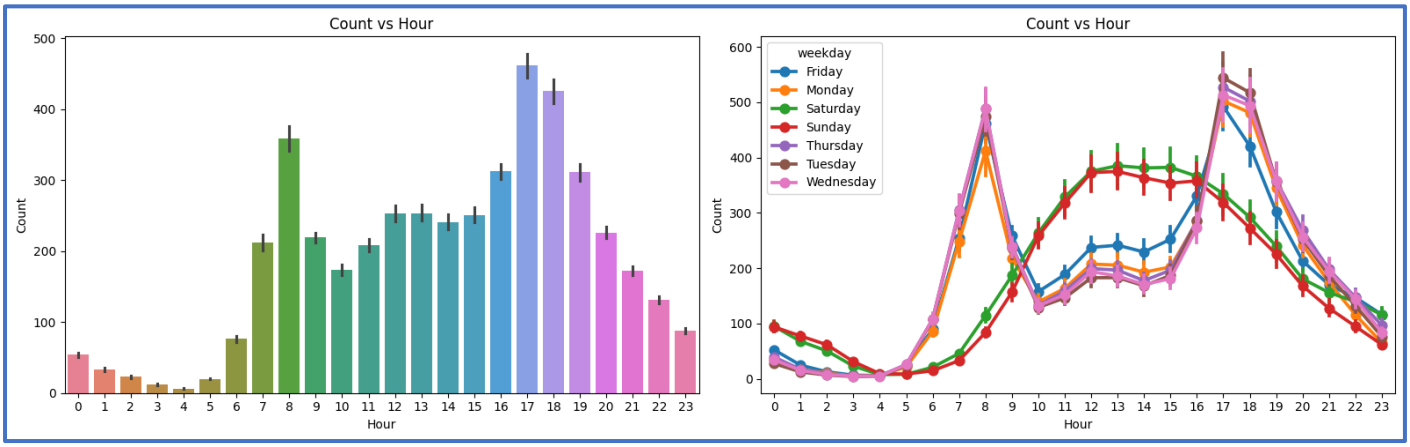

Figure 2.3: Counts vs Hour Plots (Grouped by weekdays)

- To plot the registered and casual users separately, the previous code (***Listing 2.12***) can be run after changing the y-axis to either registered or casual accordingly.
    - **Registered**:

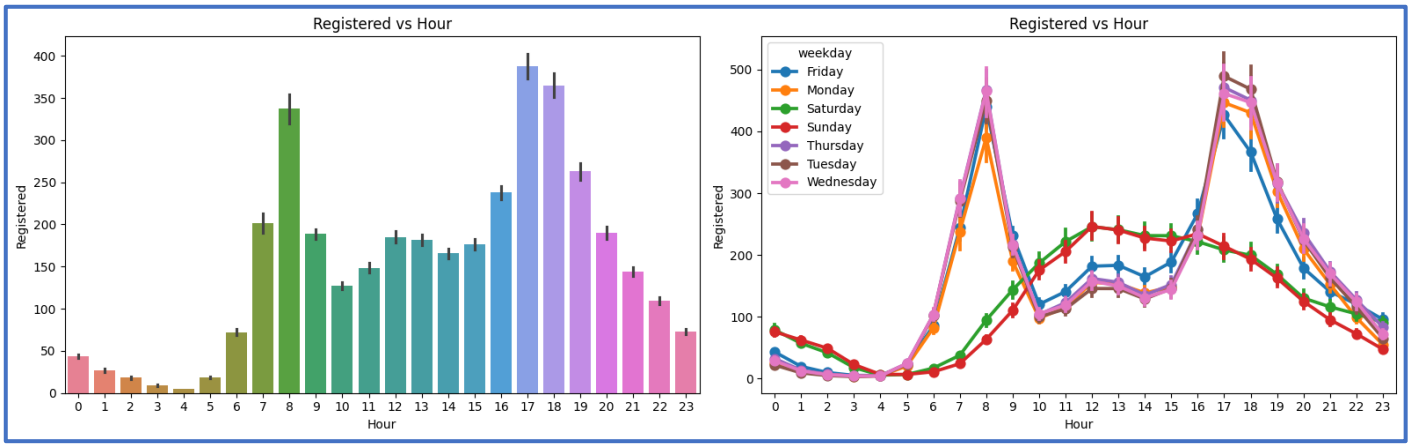

Figure 2.4: Registered vs Hour Plots (Grouped by weekdays)

    - **Casual**:

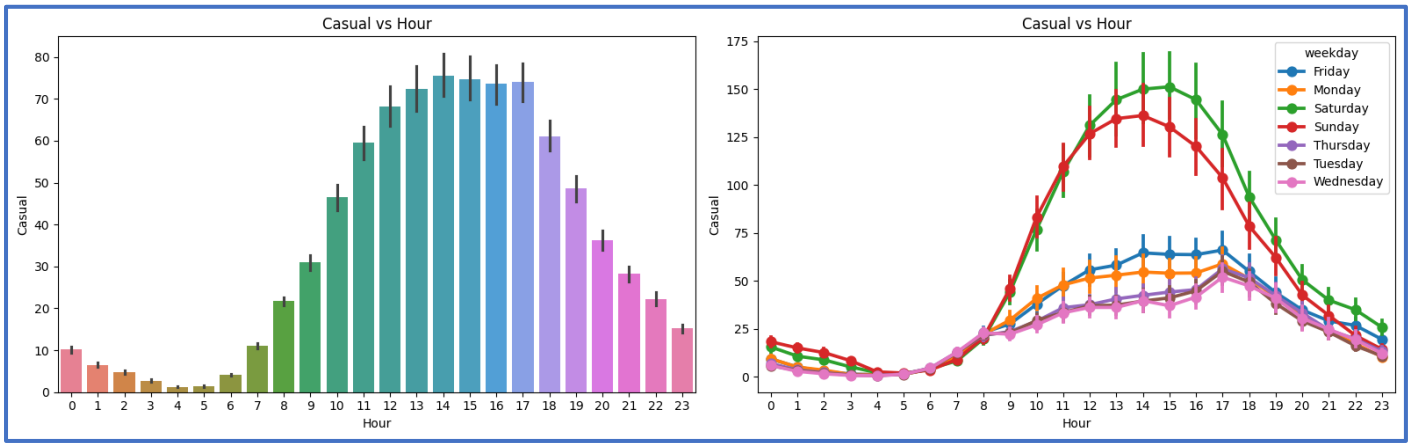

Figure 2.5: Casual vs Hour Plots (Grouped by weekdays)

The plot of registered users suggest that those users primarily use the system during peak commute hours and on working days, while still using it during mid-noon on weekends. Casual users on the other hand, primarily use the system on weekends during mid-noon, as they don't rely on it for daily usage. The combined plots of users during the weekend are what caused the bump mentioned earlier.

- To plot the total count of users per hour, grouped by different seasons and weather conditions, the following code is executed.

```
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

features = ['season', 'weather']
titles = ['Count vs Hour (during different seasons)', 'Count vs Hour (during different weather)']

for i, feature in enumerate(features):
    sns.pointplot(data=data, x='hour', y='count', hue=feature, ax=axes[i])
    axes[i].set_xlabel('Hour')
    axes[i].set_ylabel('Count')
    axes[i].set_title(titles[i])

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

Listing 2.13: Counts vs Hour Plots (Grouped by seasons, weather) Code

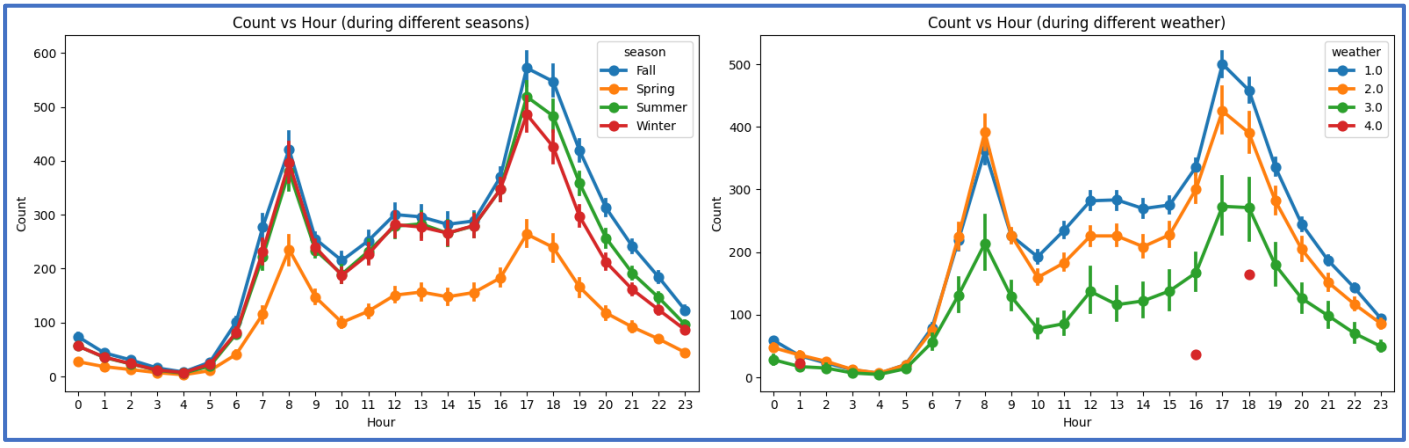

Figure 2.6: Casual vs Hour Plots (Grouped by seasons, weather)

Peak hours in fall and summer lead to higher bike rentals, while rentals decrease in spring and winter, especially in spring. Weather, also, has a substantial impact on bike rentals. Clear/partly cloudy conditions lead the highest rentals, particularly during peak hours. Misty/cloudy conditions result in somewhat decreased rentals, however light rain/snow conditions cause a significant drop in rentals. Heavy rain/snow conditions greatly reduce bike usage, indicating poor weather conditions. Therefore, understanding these weather conditions is critical for maximizing bike rentals.

2.5. Handling Missing Data and Outliers

2.5.1. Detecting Missing Data

To find the number of records with empty data, and the number of fully empty records, the following code is executed.

```
print(f"Number of records with empty data = {data.isnull().any(axis=1).sum()}")
print(f"Number of empty records = {data.isnull().all(axis=1).sum()}")
data[data.isnull().any(axis=1)].head()
```

Listing 2.14: Detecting Missing Data Code

Output:

```
Number of records with empty data = 1007
Number of empty records = 0
```

Listing 2.15: Detecting Missing Data Output

Preview of data:

Table 2.3: Preview of records with missing data

index	season	month	hour	holiday	weekday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
16	Spring	1.0	#	0.0	Saturday	0	NaN	0.42	0.4242	0.82	0.2985	41.0	52.0	93.0
17	Spring	1.0	#	0.0	Saturday	0	NaN	0.44	0.4394	0.82	0.2836	15.0	52.0	67.0
23	Spring	1.0	#	0.0	Saturday	0	NaN	0.46	0.4545	0.88	0.2985	15.0	24.0	39.0
25	Spring	1.0	1	0.0	Sunday	0	2.0	NaN	0.4394	0.94	0.2537	1.0	16.0	17.0
26	Spring	1.0	2	0.0	Sunday	0	2.0	NaN	0.4242	1.0	0.2836	1.0	8.0	9.0

There are 1007 records containing empty data, but no fully empty records. To help decide how to handle the missing data, their percentage out of the entire data should be calculated. Which could be done using the following code.

```
missing_values_percentage = (missing_values / len(data)) * 100
missing_values_percentage = missing_values_percentage[missing_values_percentage > 0]
missing_values_percentage.to_frame().T
```

Listing 2.16: Calculating Missing Data Percentages Code

Table 2.4 Percentage of Missing Data from Overall Data

Feature	month	holiday	weekday	weather	temp	atemp	humidity	windspeed	casual	registered	count
Percentage %	0,0058	0,0690	0,0058	0,0173	2,5836	0,3165	1,6859	1,7723	0,2014	0,0921	0,0690

Taking a threshold of 0.25% which equals to 43.45 of the total entries (17379 * 0.0025). Then it's possible to determine which records are safe to drop if they have missing values.

2.5.2. Handling Missing Data

2.5.2.1. Drop Records Below the Threshold

Records with missing values in the following features are safe to drop [month, holiday, weekday, weather, casual, registered, count]. The rest [temp, atemp, humidityidity, windspeed] are above the threshold so we need to find other ways to handle their missing data, such as imputation.

```
features_to_drop_records_from = ['month', 'holiday', 'weekday', 'weather', 'casual', 'registered', 'count']
data.dropna(subset=features_to_drop_records_from, inplace=True)
```

Listing 2.17: Drop Records Below Threshold Code

The remaining features are the ones above the set threshold, they need to be examined to decide how to handle their missing data. But first, the non-numeric features need to be handled to make sure the correlations can be examined correctly. That will be done later after detecting outliers.

2.5.2.2. Label-Encoding of Non-Numeric Features on a Copy of the Data

Label encoding needs to be done on the non-numeric features to proceed with handling missing values, in order to find the correlation between features.

```
label_encoder = LabelEncoder()
data_copy = data.copy()
data copy['season'] = label encoder.fit transform(data copy['season'])
data_copy['weekday'] = label_encoder.fit_transform(data_copy['weekday'])

data_copy.head()
```

Listing 2.18: Label Encode Non-Numeric Features Code

Table 2.5: Data after Label Encoding Non-Numeric features

index	season	month	hour	holiday	weekday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	1	1	0	0	2	0	1	0,24	0,2879	0,81	0	3	13	16
1	1	1	1	0	2	0	1	0,22	0,2727	0,8	0	8	32	40
2	1	1	2	0	2	0	1	0,22	0,2727	0,8	0	5	27	32
3	1	1	3	0	2	0	1	0,24	0,2879	0,75	0	3	10	13
4	1	1	4	0	2	0	1	0,24	0,2879	0,75	0	0	1	1

Now, the correlation between features can be found to examine which features correlate most with each other.

2.5.2.3. Correlation Matrix

To find the correlation between the features, the following code was executed.

```
plt.figure(figsize=(15, 7))
sns.heatmap(data_copy.corr(), annot=True, cmap='BuPu', annot_kws={'size':10})
plt.title('Correlation Matrix')
plt.show()
```

Listing 2.19: Correlation Matrix Code

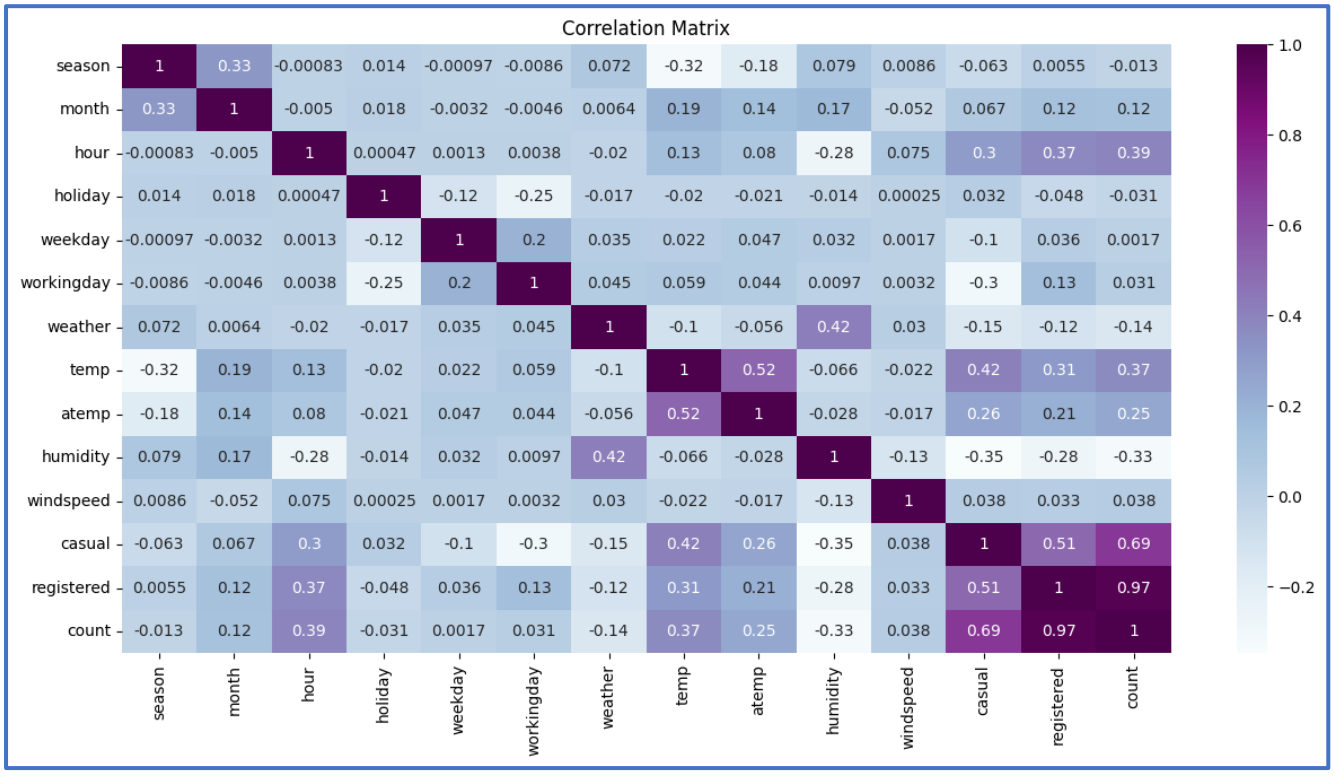

Figure 2.7: Correlation Matrix

The features 'temp' and 'atemp' have the highest correlation together. The feautre 'humidity' correlates the highest with 'weather'. Feature 'windspeed' correlates the highest with 'humidity'.

2.5.2.4. Missing Data Imputation

Using the KNN technique, the missing values can be imputed based on the correlation between the features.

```
features_for_imputation = ['temp', 'atemp', 'humidity', 'windspeed']
data_for_imputation = data[features_for_imputation]
data = data.copy()

knn_imputer = KNNImputer(n_neighbors=5, weights='uniform', metric='nan_euclidean')
data_imputed = knn_imputer.fit_transform(data_for_imputation)
data_imputed_df = pd.DataFrame(data_imputed, columns=features_for_imputation,
index=data_for_imputation.index)
data[features_for_imputation] = data_imputed_df[features_for_imputation]

print(data.isnull().sum())
```

Listing 2.20: Missing Values Imputation Code

Table 2.6: Number of Missing Values after Handling All Missing Data

Feature	month	holiday	weekday	weather	temp	atemp	humidity	windspeed	casual	registered	count
# Missing Values	0	0	0	0	0	0	0	0	0	0	0

Now that all missing values are handled, the non-scaled numeric features can be scaled to allow for outliers' detection.

2.5.3. Scaling Numerical Features

```
features = ['casual', 'registered', 'count']
scaler = MinMaxScaler()
data[features] = scaler.fit_transform(data[features])
data.describe()
```

Listing 2.21: Feature Scaling Code

Table 2.7: Numeric Data Stats After Scaling

stat	temp	atemp	humidity	windspeed	casual	registered	count
count	17300,0000	17300,0000	17300,0000	17300,0000	17300,0000	17300,0000	17300,0000
mean	0,4988	0,4811	0,6272	0,1923	0,0973	0,1738	0,1933
std	0,2104	0,2899	0,1919	0,2413	0,1343	0,1709	0,1859
min	0,0200	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
25%	0,3400	0,3333	0,4800	0,1045	0,0109	0,0384	0,0400
50%	0,5000	0,4848	0,6300	0,1881	0,0463	0,1309	0,1445
75%	0,6600	0,6212	0,7800	0,2537	0,1308	0,2483	0,2869
max	7,0000	14,0000	1,0000	17,0000	1,0000	1,0000	1,0000

After scaling, the ranges of the numeric features are the same. However, some features are showing max values above the range, this is due to them having outliers. Which will be detected and handled next.

2.5.4. Detecting Outliers

2.5.4.1. Using Boxplot

- To detect ‘count’ outliers using Boxplot the following code is executed.

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(16, 5))

data.boxplot(by='month', column='count', grid=False, ax=axes[0])
axes[0].set_title('Boxplot of count by month')
axes[0].set_xlabel('Month')
axes[0].set_ylabel('Count')

data.boxplot(by='weekday', column='count', grid=False, ax=axes[1])
axes[1].set_title('Boxplot of count by weekday')
axes[1].set_xlabel('Weekday')
axes[1].set_ylabel('Count')

data.boxplot(column='casual', grid=False, ax=axes[2])
axes[2].set_title('Boxplot of count by season')
axes[2].set_xlabel('Season')
axes[2].set_ylabel('Count')

fig.suptitle('')
plt.tight_layout()
plt.show()
```

Listing 2.22: Count Boxplots Code

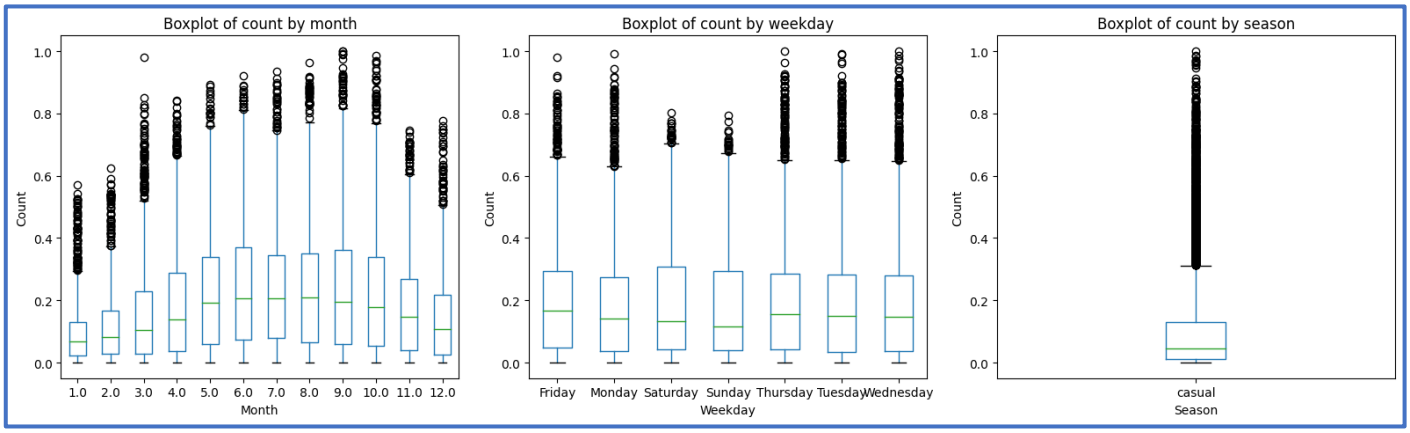

Figure 2.8: Count Boxplots

The plots above indicate that the count feature has lots of outliers, and can be visualized by grouping them with different features.

- To detect outliers for the rest of the continuous features using Boxplot the following code is executed.

```
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(14, 8))

# List of features to plot
features = ['temp', 'atemp', 'humidity', 'windspeed', 'casual', 'registered']

# Create boxplots for each feature
for i, feature in enumerate(features):
    row, col = divmod(i, 3)
    data.boxplot(column=feature, grid=False, ax=axes[row, col])
    axes[row, col].set_title(f'Boxplot of {feature}')
    axes[row, col].set_ylabel('Value')

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

Listing 2.23: Features Boxplots Code

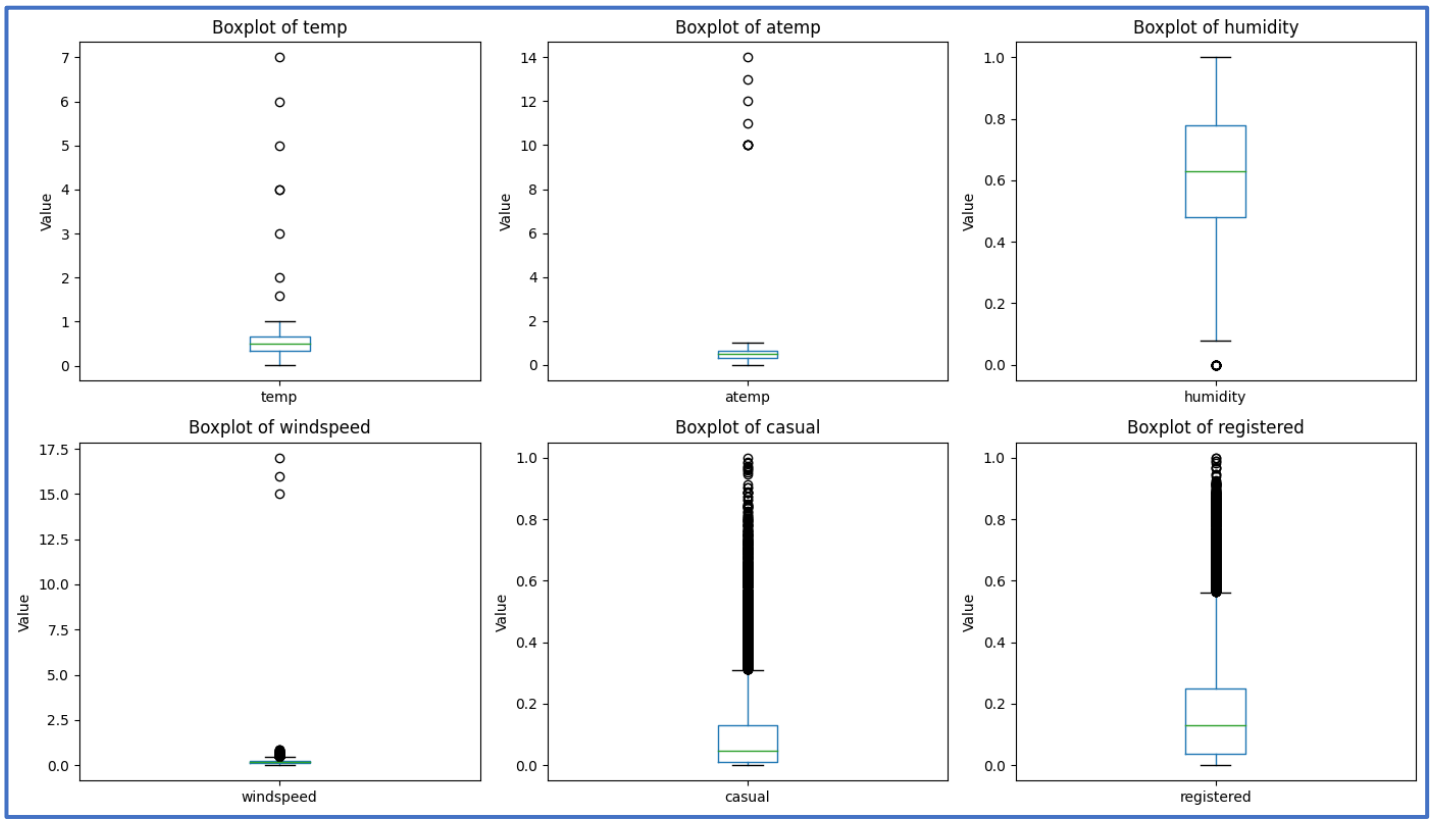

Figure 2.9: Features Boxplots

The plots above are the features that have outliers which definitely need handling. The outliers vary for each feature.

2.5.4.2. Calculating IQR and Number of Outliers

Calculating the IQR for the data helps better understand the nature of the outliers and how to handle them. To do so, the following code is executed.

```
features = ['temp', 'atemp', 'humidity', 'windspeed', 'casual', 'registered', 'count']

percentile_25 = data[features].quantile(0.25)
percentile_50 = data[features].quantile(0.5)
percentile_75 = data[features].quantile(0.75)

iqr = percentile_75 - percentile_25

lower_bound = percentile_25 - 1.5*iqr
upper_bound = percentile_75 + 1.5*iqr

summary_table = pd.DataFrame({
    '25th Percentile': percentile_25,
    '50th Percentile': percentile_50,
    '75th Percentile': percentile_75,
    'IQR': iqr,
    'Lower Bound': lower_bound,
    'Upper Bound': upper_bound
})

summary_table.T
```

Listing 2.24: Calculating IQR Code

Table 2.8: IQR Summary

	temp	atemp	humidity	windspeed	casual	registered	count
25th Percentile	0,3400	0,3333	0,4800	0,1045	0,0109	0,0384	0,0400
50th Percentile	0,5000	0,4848	0,6300	0,1881	0,0463	0,1309	0,1445
75th Percentile	0,6600	0,6212	0,7800	0,2537	0,1308	0,2483	0,2869
IQR	0,3200	0,2879	0,3000	0,1492	0,1199	0,2099	0,2469
Lower Bound	-0,1400	-0,0986	0,0300	-0,1193	-0,1689	-0,2765	-0,3304
Upper Bound	1,1400	1,0531	1,2300	0,4775	0,3106	0,5632	0,6573

To find the percentage of outliers from the entire data, the following code is executed.

```
outlier_info = {
    "# Outliers": [],
    "Max Outlier": [],
    "Min Outlier": [],
    "% of Outliers": []
}

for feature in features:
    outliers = data[(data[feature] < lower_bound[feature]) | (data[feature] > upper_bound[feature])]
    outlier_info["# Outliers"].append(len(outliers))
    outlier_info["Max Outlier"].append(outliers[feature].max())
    outlier_info["Min Outlier"].append(outliers[feature].min())
    outlier_info["% of Outliers"].append(100 * len(outliers) / len(data))

outlier_info_df = pd.DataFrame(outlier_info, index=features)
outlier_info_df.T
```

Listing 2.25: Outlier Percentage Code

Table 2.9: Outliers Percentages

index	temp	atemp	humidity	windspeed	casual	registered	count
# Outliers	8,0000	8,0000	22,0000	338,0000	1187,0000	679,0000	504,0000
Max Outlier	7,0000	14,0000	0,0000	17,0000	1,0000	1,0000	1,0000
Min Outlier	1,6000	10,0000	0,0000	0,4836	0,3134	0,5643	0,6578
% of Outliers	0,0462	0,0462	0,1272	1,9538	6,8613	3,9249	2,9133

2.5.5. Handling Outliers

Taking a threshold of 1.0% (173 of total records) for the outliers, the outliers in [temp, atemp, humidity] can be removed as they have very little effect on the data. While the rest [windspeed, casual, registered, count] are above the threshold and must have considerable effect on the data. Capping will be done to them.

- To delete outliers from [temp, atemp, humidity], the following code is executed.

```
filtered_data = data.copy()
features_for_dropping = ['temp', 'atemp', 'humidity']

for feature in features_for_dropping:
    filtered_data = filtered_data[(filtered_data[feature] >= lower_bound[feature]) &
(filtered_data[feature] <= upper_bound[feature])]

print(filtered_data.shape)
print(f"Original dataset size: {len(data)}")
print(f"Cleaned dataset size: {len(filtered_data)}")
```

Listing 2.26: Deleting Outliers Code

```
(17262, 14)
Original dataset size: 17300
Cleaned dataset size: 17262
```

Listing 2.27: Original Dataset Length vs Cleaned Dataset Length

- To delete outliers from [temp, atemp, humidity], the following code is executed.

```
for feature in ['windspeed', 'registered', 'count']:
    cap_value = filtered_data[feature].quantile(0.95)
    floor_value = filtered_data[feature].quantile(0.05)
    filtered_data[feature] = np.where(filtered_data[feature] > cap_value, cap_value,
filtered_data[feature])
    filtered_data[feature] = np.where(filtered_data[feature] < floor_value, floor_value,
filtered_data[feature])

cap_value = upper_bound['casual']
floor_value = data['casual'].quantile(0.05)
filtered_data['casual'] = np.where(filtered_data['casual'] > cap_value, cap_value,
filtered_data['casual'])
filtered_data['casual'] = np.where(filtered_data['casual'] < floor_value, floor_value,
filtered_data['casual'])

for feature in features:
  outliers = filtered data[((filtered data[feature] < lower bound[feature]) | (filtered data[feature] >
upper_bound[feature]))]
  print(f"Number of outliers in {feature}: {len(outliers)}")
```

Listing 2.28: Capping Outliers Code

```
Number of outliers in temp: 0
Number of outliers in atemp: 0
Number of outliers in humidity: 0
Number of outliers in windspeed: 0
Number of outliers in casual: 0
Number of outliers in registered: 0
Number of outliers in count: 0
```

Listing 2.29: Number of Outliers After Handling

2.6. Visualizing Distributions

```
fig, axes = plt.subplots(3, 3, figsize=(16, 15))

features = ['registered', 'casual', 'temp', 'atemp', 'humidity', 'windspeed', 'count']
titles = ['Count vs Registered', 'Count vs Casual', 'Count vs Temp', 'Count vs Atemp', 'Count vs
Humidity', 'Count vs Windspeed', 'Count vs Count']
colors = ['plum', 'teal', 'green', 'purple', 'orange', 'crimson', 'steelblue']

for ax, feature, title, color in zip(axes.flat, features, titles, colors):
    sns.histplot(data=filtered_data, x=feature, ax=ax, kde=True, color=color)
    ax.set_xlabel(feature.capitalize())
    ax.set_title(title)

for i in range(len(features), len(axes.flat)):
    fig.delaxes(axes.flat[i])

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

Listing 2.30: Features Boxplots Code

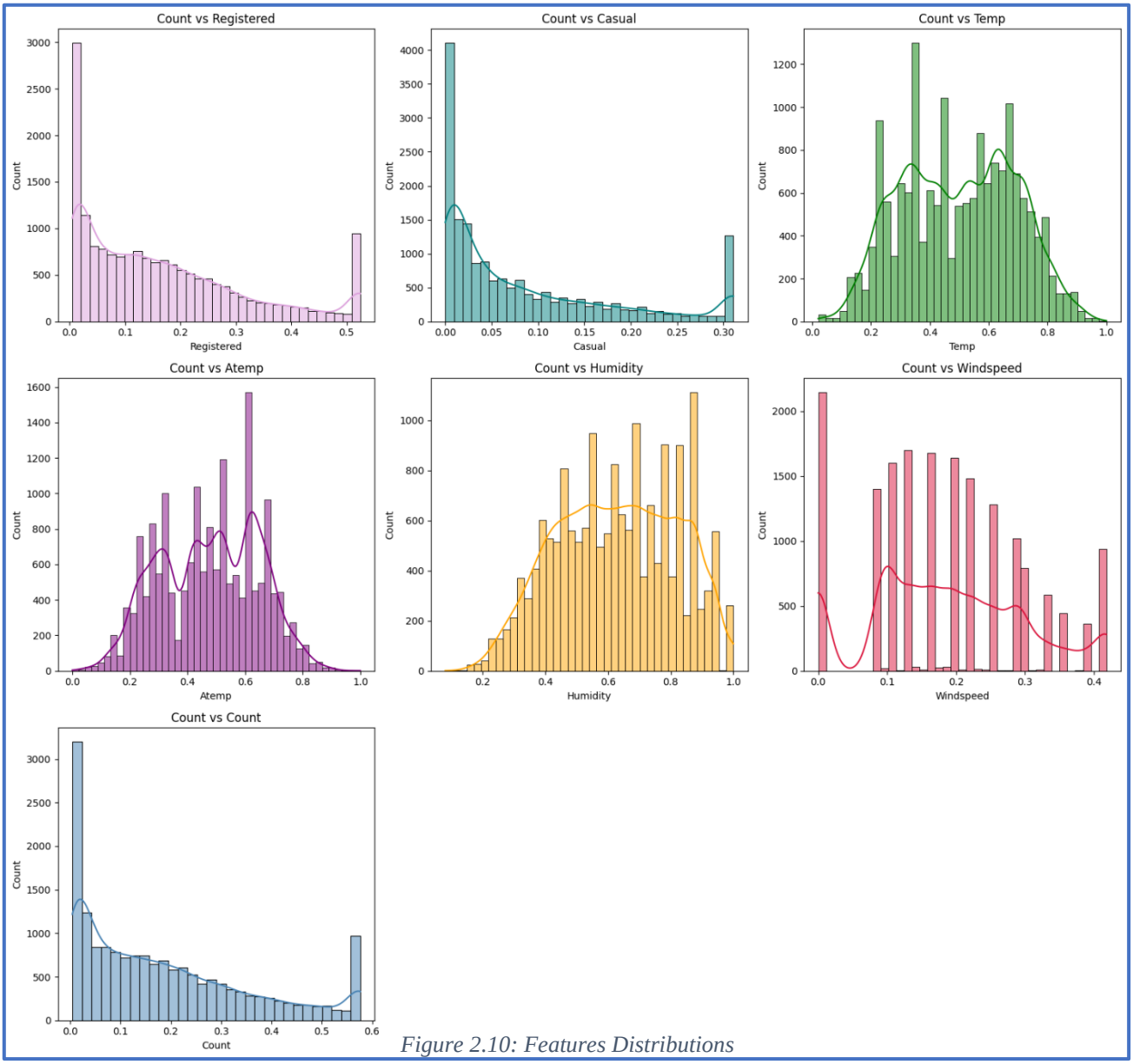

Figure 2.10: Features Distributions

The previous distributions (Figure 2.10) reveal key patterns in bike rental behavior influenced by user type and weather conditions. Both registered and casual users show a skew toward lower usage, with registered users having a longer tail, indicating the presence of some high-usage individuals. Moderate and mild temperatures are correlated with higher bike rental numbers, implying that weather comfort plays a key role in bike usage. In contrast, extreme temperatures, high humidity, and strong winds have a negative effect on bike rentals, resulting in decreased usage during these conditions.

2.7. Encoding Categorical Data Using One-Hot Encoding

Encoding Categorical features using One-Hot Encoding helps in the process of feature selection and dimensionality reduction, by deciding which columns are most important and which aren't.

```
categorical_columns1 = ['season', 'weekday']
categorical_columns2 = ['month', 'hour', 'holiday', 'workingday', 'weather']
data_encoded = filtered_data.copy()

for feature in categorical_columns2:
  enc = OneHotEncoder(handle_unknown='ignore', sparse=False)
  df_feature = enc.fit_transform(data_encoded[[feature]])
  feature_names = enc.get_feature_names_out([feature])
  df_feature_df = pd.DataFrame(df_feature, columns=feature_names)
  df_feature_df.reset_index(drop=True, inplace=True)
  data_encoded = pd.concat([data_encoded.reset_index(drop=True), df_feature_df], axis=1)

for feature in categorical_columns1:
  enc = OneHotEncoder(handle_unknown='ignore')
  enc.fit(data_encoded[[feature]])
  df_feature = enc.transform(data_encoded[[feature]]).toarray()
  data_encoded[enc.categories_[0]] = df_feature

data_encoded = data_encoded.drop(columns=categorical_columns1)
data_encoded = data_encoded.drop(columns=categorical_columns2)
data_encoded.columns
```

Listing 2.31: 2.7. Encoding Categorical Data Code

```
Index(['temp', 'atemp', 'humidity', 'windspeed', 'casual', 'registered',
       'count', 'month_1.0', 'month_2.0', 'month_3.0', 'month_4.0',
       'month_5.0', 'month_6.0', 'month_7.0', 'month_8.0', 'month_9.0',
       'month_10.0', 'month_11.0', 'month_12.0', 'hour_0', 'hour_1', 'hour_2',
       'hour_3', 'hour_4', 'hour_5', 'hour_6', 'hour_7', 'hour_8', 'hour_9',
       'hour_10', 'hour_11', 'hour_12', 'hour_13', 'hour_14', 'hour_15',
       'hour_16', 'hour_17', 'hour_18', 'hour_19', 'hour_20', 'hour_21',
       'hour_22', 'hour_23', 'holiday_0.0', 'holiday_1.0', 'workingday_0',
       'workingday_1', 'weather_1.0', 'weather_2.0', 'weather_3.0',
       'weather_4.0', 'Fall', 'Spring', 'Summer', 'Winter', 'Friday', 'Monday',
       'Saturday', 'Sunday', 'Thursday', 'Tuesday', 'Wednesday'],
dtype='object')
```

Listing 2.32: 2.7. Encoding Categorical Data Output

Now the data consists of 61 columns.

2.8. Feature Selection

A good starting point could be selecting around 10-30% of the total number of features.

2.8.1. Finding the Best Number of Features to Retain Using Cross Validation

```
# Separate features and target
X = data_encoded.drop(columns=['count'])
y = data_encoded['count']
k_values = range(5, 30, 5)
scores = []

for k in k_values:
    selector = SelectKBest(score_func=mutual_info_regression, k=k)
    X_selected = selector.fit_transform(X, y)
    model = LinearRegression()
    score = cross_val_score(model, X_selected, y, cv=5, scoring='neg_mean_squared_error')
    scores.append(score.mean())

plt.plot(k_values, scores)
plt.xlabel('Number of Features (k)')
plt.ylabel('Cross-Validated MSE')
plt.title('Feature Selection using SelectKBest')
plt.show()

best_k = k_values[scores.index(max(scores))]
print(f'The best number of features (k) is: {best_k}')
```

Listing 2.33: SelectKBest Cross Validation Code

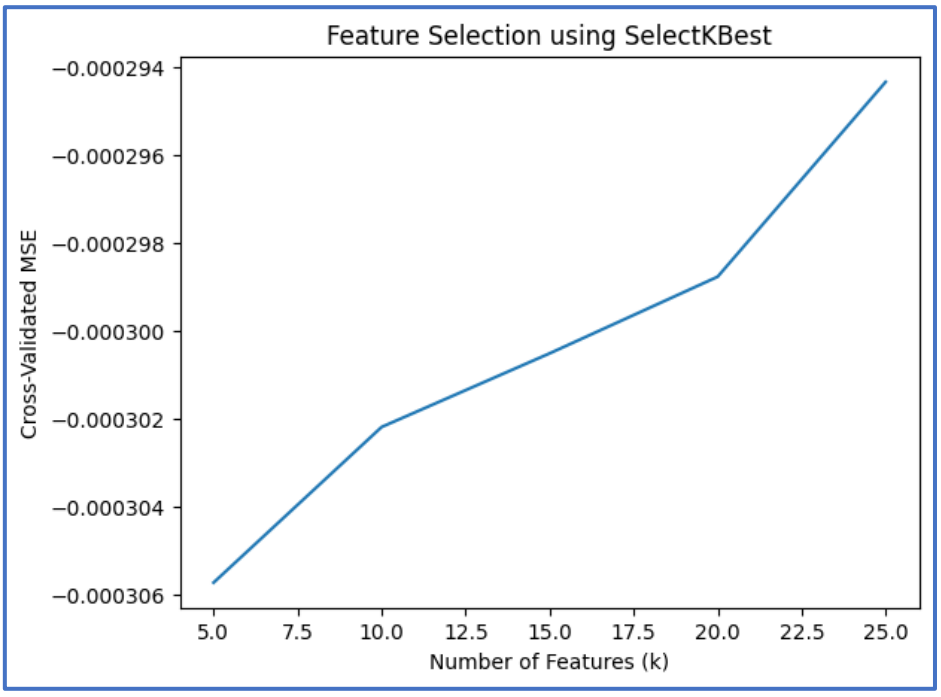

Figure 2.11: : SelectKBest Cross Validation

From the graph above, it can be found that the best number of features (k) is 25.

2.8.2. Performing Feature Selection

Feature Selection will be performed to retain 25 features.

```
selector = SelectKBest(score_func=mutual_info_regression, k=25)
selector.fit(X, y)

# Get the selected features
selected_features = selector.get_support(indices=True)
selected feature names = X.columns[selected features]

# Transform the dataset to contain only the selected features
X_selected = selector.transform(X)



print("Selected features:")
print(selected feature names)
```

Listing 2.34: Performing Feature Selection Code

```
Selected features:
Index(['temp', 'atemp', 'humidity', 'windspeed', 'casual', 'registered',
       'hour_0', 'hour_1', 'hour_2', 'hour_3', 'hour_4', 'hour_5', 'hour_6',
       'hour_9', 'hour_14', 'hour_16', 'hour_17', 'hour_18', 'hour_20',
       'hour_22', 'hour_23', 'weather_3.0', 'Fall', 'Spring', 'Winter'],
      dtype='object')
```

Listing 2.35: Selected Features

2.9. Dimensionality Reduction Using PCA

2.9.1. Finding the Best Number of Components to Retain

To determine the best number of components to retain out 25, which should be around 90-95% of the total number. The number of components against the cumulative variance will be plotted, then the point where the plot starts to flatten will be the best number.

```
X_scaled = X_selected_df.copy()
pca = PCA().fit(X_scaled)

# Plot the cumulative explained variance
plt.figure(figsize=(14, 6))
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance by PCA Components')
plt.xticks(range(1, len(np.cumsum(pca.explained_variance_ratio_)) + 1))
plt.grid()
plt.show()
```

Listing 2.36: Finding the Best Number of Components to Retain Code

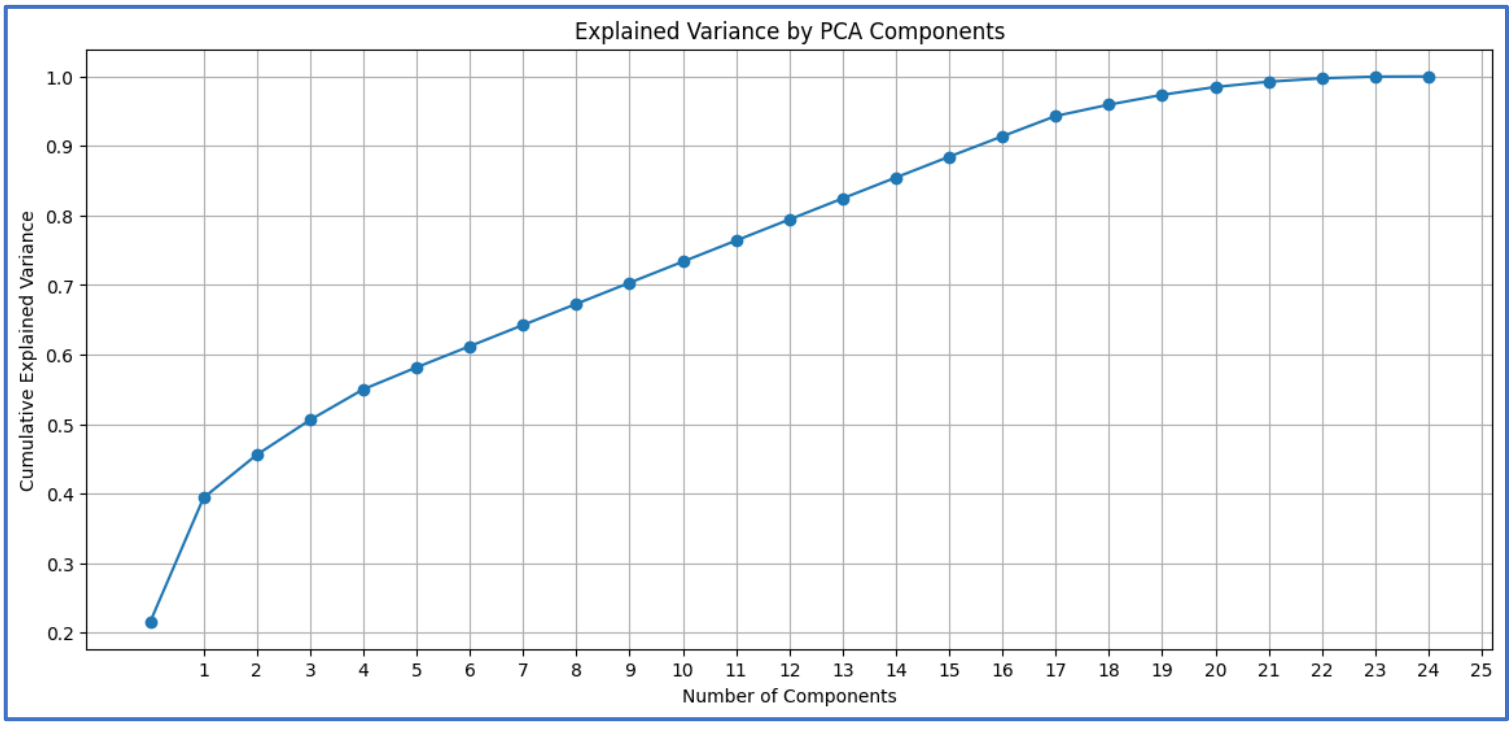

Figure 2.12: Explained Variance by PCA Components

The graph shows that the plot starts to flatten out around 23, which is 92% out of 25 (total number).

2.9.2. Performing PCA

```
n_components = 23
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_scaled)
explained_variance = pca.explained_variance_ratio_

fig, ax = plt.subplots(1, 2, figsize=(15, 6))
# individual explained variance
ax[0].bar(range(1, n_components + 1), explained_variance, alpha=0.7, color='blue',
align='center')
ax[0].set xlabel('Principal Components')
ax[0].set_ylabel('Explained Variance Ratio')
ax[0].set_title('PCA - Individual Explained Variance')
ax[0].set xticks(range(1, n components + 1))
# cumulative explained variance
ax[1].plot(range(1, n_components + 1), np.cumsum(explained_variance), marker='o', color='red')
ax[1].set_xlabel('Principal Components')
ax[1].set_ylabel('Cumulative Explained Variance')
ax[1].set_title('PCA - Cumulative Explained Variance')
ax[1].set_xticks(range(1, n_components + 1))

plt.tight_layout()
plt.show()

cumulative_variance = np.cumsum(explained_variance)
print(f'Cumulative explained variance for {n_components} components: {cumulative_variance[-1]}')
```

Listing 2.37: Performing PCA and Plotting Variance Code

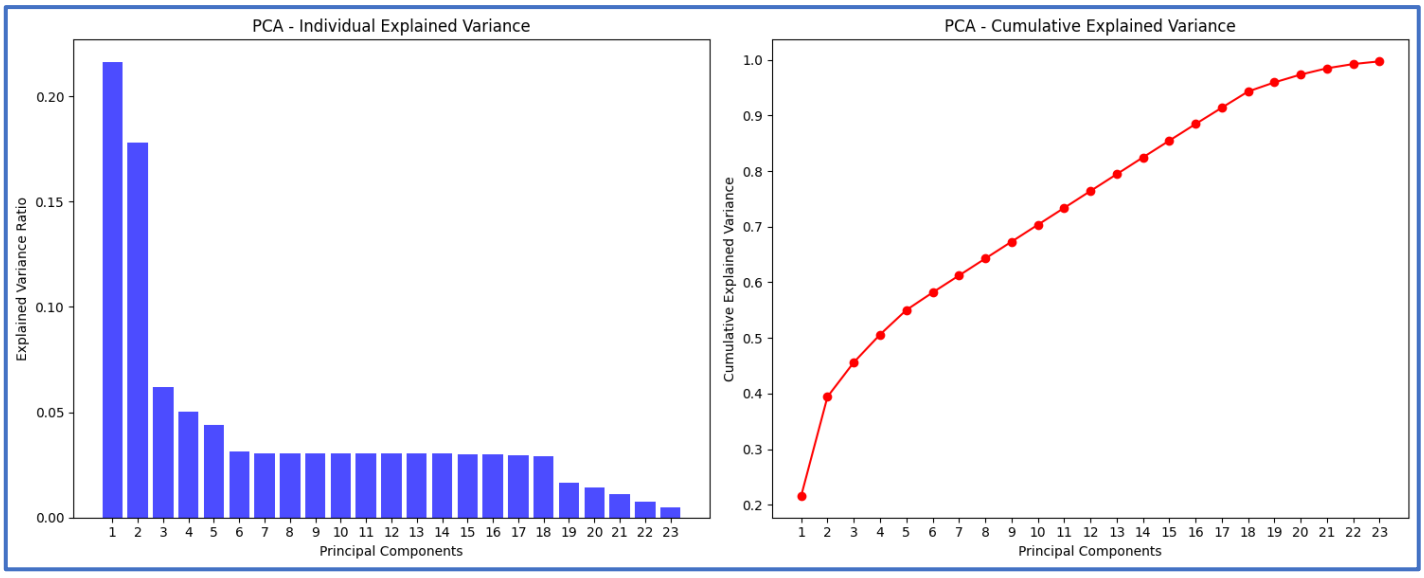

Figure 2.13: Plots of Individual and Cumulative Variance

Cumulative explained variance for 23 components: 0.9972697874025593

From the plot, the most informative components are retained now.

2.10. Data After Feature Selection and PCA

```
# Create a new DataFrame 'df_PCA' from the transformed data
data_PCA = pd.DataFrame(data=X_pca, columns=[f'PC{i+1}' for i in range(n_components)])
data_PCA.reset_index(drop=True, inplace=True)
data_PCA.head()
data_PCA.info()
```

Listing 2.38: DataFrame After Cleaning Data

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11	PC12	PC13	PC14	PC15	PC16	PC17	PC18	PC19	PC20	PC21	PC22	PC23
0	-0.675272	-0.568114	0.117595	-0.286859	-0.128780	0.078787	0.048430	-0.046292	-0.020754	0.083130	-0.344941	-0.592633	-0.503416	-0.266358	-0.252104	-0.013767	-0.135616	-0.027416	0.206818	0.018207	-0.003708	0.121614	-0.029636
1	-0.684187	-0.568576	0.124386	-0.298054	-0.139993	0.080734	0.045904	-0.059127	-0.012142	0.029471	-0.318410	0.751661	-0.072969	-0.305445	-0.276732	-0.020283	-0.148293	-0.030399	0.185425	0.009413	0.011090	0.130895	-0.066578
2	-0.684585	-0.567740	0.136885	-0.303655	-0.149724	0.044583	0.008277	-0.017297	-0.001319	-0.003933	-0.019914	0.001364	-0.004739	0.075291	0.598393	-0.663202	-0.260603	-0.046588	0.175952	-0.002310	0.001048	0.129913	-0.071254
3	-0.674906	-0.569269	0.121736	-0.298902	-0.151190	0.022802	0.002324	-0.006806	-0.000795	-0.001289	-0.006787	0.000023	-0.000667	0.023594	0.087309	0.017249	0.732709	-0.579619	0.128289	-0.026069	-0.039250	0.141476	-0.065906
4	-0.675943	-0.568283	0.133793	-0.307492	-0.155644	0.006248	0.000834	-0.008402	-0.000147	-0.006716	-0.002726	0.000811	-0.004182	0.023962	0.077604	0.019006	0.463090	0.808257	0.116373	-0.036452	-0.051841	0.151036	-0.063413

Figure 2.14: Preview of Cleaned DataFrame

Cleaned data info:

```
RangeIndex: 17262 entries, 0 to 17261
Data columns (total 23 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   PC1     17262 non-null  float64
 1   PC2     17262 non-null  float64
 2   PC3     17262 non-null  float64
 3   PC4     17262 non-null  float64
 4   PC5     17262 non-null  float64
 5   PC6     17262 non-null  float64
 6   PC7     17262 non-null  float64
 7   PC8     17262 non-null  float64
 8   PC9     17262 non-null  float64
 9   PC10    17262 non-null  float64
 10  PC11    17262 non-null  float64
 11  PC12    17262 non-null  float64
 12  PC13    17262 non-null  float64
 13  PC14    17262 non-null  float64
 14  PC15    17262 non-null  float64
 15  PC16    17262 non-null  float64
 16  PC17    17262 non-null  float64
 17  PC18    17262 non-null  float64
 18  PC19    17262 non-null  float64
 19  PC20    17262 non-null  float64
 20  PC21    17262 non-null  float64
 21  PC22    17262 non-null  float64
 22  PC23    17262 non-null  float64
dtypes: float64(23)
```

Listing 2.39: Cleaned Data Info

2.11. Training Model on Data

Training and testing will be conducted for the original data, the data after feature selection, and the final cleaned data after PCA. All three will be split into training and testing data, and their results will be compared.

```
X train, X test, y train, y test = train test split(X, y, test size=0.2, random state=42)
X_filtered_train, X_filtered_test, y_filtered_train, y_filtered_test =
train_test_split(X_selected_df, y, test_size=0.2, random_state=42)
X_pca_train, X_pca_test, y_pca_train, y_pca_test = train_test_split(X_pca, y, test_size=0.2,
random_state=42)

# Train a regressor on the original features and evaluate
regressor = RandomForestRegressor(random_state=42)
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_test)
r2_original = r2_score(y_test, y_pred)

# Train a regressor on the selected features and evaluate
regressor.fit(X_filtered_train, y_filtered_train)
y_pred_filtered = regressor.predict(X_filtered_test)
r2_filtered = r2_score(y_filtered_test, y_pred_filtered)

# Train a regressor on the PCA-transformed features and evaluate
regressor.fit(X_pca_train, y_pca_train)
y_pred_pca = regressor.predict(X_pca_test)
r2_pca = r2_score(y_pca_test, y_pred_pca)

print(f"Number of original features: {X_train.shape[1]}")
print(f"Number of features after feature selection: {X_filtered_train.shape[1]}")
print(f"Number of features after PCA: {X_pca_train.shape[1]}")
print(f"R-squared of Original features (testing accuracy): {r2_original}")
print(f"R-squared after feature selection (testing accuracy): {r2_filtered}")
print(f"R-squared after PCA (testing accuracy): {r2_pca}")
```

Listing 2.40: Training and Testing All Data

```
Number of original features: 61
Number of features after feature selection: 25
Number of features after PCA: 23
R-squared of Original features (testing accuracy): 0.996840633467523
R-squared after feature selection (testing accuracy): 0.9960960679922346
R-squared after PCA (testing accuracy): 0.9882678900144292
```

Listing 2.41: Model Learning Comparison

The original features model demonstrated a strong fit, explaining **99.68%** of the variance in the target feature ‘count’, indicating a very strong fit. **The selected features model**, on the other hand, explained **99.61%** of the variance, indicating a high level of explanation despite a very slight drop (0.07%) compared to the original feature model. Finally, **the PCA-transformed features model** explained **98.82%** of the variance, also indicating a high level of explanation despite a slight drop (0.1986%) compared to the original feature model.

Using feature selection and PCA has reduced the dimensionality of the dataset significantly (from 61 features to 23) while retaining a substantial amount of the information (variance) in the data. This means all the preprocessing work done before is valid and contributes to building a good learning model.

3. Conclusion

The comprehensive data cleaning and feature engineering efforts undertaken in this report significantly contribute to the preparation of the Bike Sharing Dataset for predictive modeling. By addressing missing values, outliers, and appropriately encoding categorical variables, the dataset is rendered more robust and reliable for analysis. Visualization of the data reveals critical insights, such as the influence of weather conditions and time of day on bike rental patterns.

The initial model on the raw dataset had a testing accuracy of 99.68%. The feature selection process identified key attributes like temperature, humidity, and wind speed as significant predictors of bike rentals. Using SelectKBest for feature selection we were able to streamline the dataset while retaining 99.6% of the variance in the data, ensuring that the most relevant information is used for model training. The performance metric R-squared was employed to evaluate the predictive models. The final model achieved an R-squared value of 0.988, indicating that 98.8% of the variability in bike rental counts can be explained by the model.

These steps collectively enhance the understanding of the factors driving bike sharing usage and improve the potential for accurate predictive models. Ultimately, this supports the development of efficient and user-friendly bike sharing systems, capable of adapting to various environmental conditions and user behaviors.