BIRZEIT UNIVERSITY

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

Case Study #1: Data Cleaning and Feature Engineering for the Bike Sharing Dataset

Prepared by: Nafe Abbaker—1200047

Instructor: **Dr.Mohammad Jubran**

Assistant: **Eng. Hanan**

Date: **August,2,2024**

Abstract

This report explores the application of machine learning techniques to predict bike rental demand using historical data from the Capital Bikeshare system in Washington D.C. The study focuses on data from 2011-2012 and employs Decision Tree and Random Forest Regressors to model rental counts. A comprehensive data cleaning and feature engineering process was undertaken, including handling missing data, encoding categorical variables, and identifying outliers. Key features influencing bike rentals, such as temporal factors and weather conditions, were identified and analyzed. Dimensionality reduction was performed using Principal Component Analysis (PCA) to enhance model performance. The results indicate that temporal features, such as the hour of the day, play a crucial role in predicting rental demand. The study concludes that preprocessing and feature selection significantly improve model accuracy and suggests that further exploration of advanced models and time-series analyses could provide additional insights for optimizing bike-sharing systems.

Table of Contents

1 Introduction

Bike-sharing systems play a critical role in urban mobility planning and optimization. The challenge of predicting bike rental demand arises from its reliance on numerous factors, including weather conditions, urban infrastructure, and temporal variables. Effective prediction models can aid city planners and service providers in managing inventory and improving service quality. This project utilizes data from the Capital Bikeshare system, focusing on the years 2011-2012, to explore how different variables affect bike rental patterns. By employing a Decision Tree Regressor, the study investigates the predictive power of various features, aiming to improve the understanding and forecasting of bike rental demand.

2 Procedure

2.1 Libraries Importing

```
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.tree import plot_tree
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.tree import DecisionTreeRegressor

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Figure 1 Libraries Used

Here's a concise summary of the rationale behind the usage of specific Python libraries in the analysis of the bike-sharing dataset (hours.csv):

1. **Pandas**: Used for loading the CSV file and performing various data manipulation tasks such as handling missing values, filtering, and transforming data.
2. **NumPy**: Employed for efficient numerical operations on arrays and matrices, crucial for data calculations.
3. **Matplotlib**: Utilized for creating a wide range of static and interactive plots to visualize data distributions and model performance.
4. **Seaborn**: Provides a high-level interface for drawing statistical graphics, making it easier to generate complex visualizations with less code.
5. **Scikit-learn**:
    - **Preprocessing modules**: These prepare the dataset for modeling through encoding, scaling, and handling missing values.
    - **Model selection tools**: Used to split the dataset and optimize model parameters.
    - **Machine learning models**: Decision trees and random forests were used for predictive modeling due to their effectiveness with non-linear data.
    - **Evaluation metrics**: Employed to assess model performance, providing insights intoaccuracy, precision, recall, and error metrics.

2.2 Data Set Loading and exploring it

Then we loaded the data and started to work with it as below

Loading Dataset

Loading bike sharing dataset

```
[2] df = pd.read_csv('/content/hours.csv')
```

Figure 2 DataSet Loading

For more information about the dataset, we used some functions that gives us more understanding of what information the dataset involves.

Exploring dataset

Analaysing the dataset, knowing more information about it.

Printing some values important to undetstand the dataset

```
[37] print("df.head()\n")
     print(df.head())
     print("---------------------------------------------------------\n")
     print("df.info()\n")
     print(df.info())
     print("---------------------------------------------------------\n")
     print("df.describe()\n")
     print(df.describe())
     print("---------------------------------------------------------\n")
     # df.tail()
```

Figure 3 Explring dataset

I chose df.head() to show the first 5 rows of the dataset, output is shown below

```
df.head()

   instant    dteday  season  yr  mnth  hr  holiday   weekday  workingday  \
0        1  1/1/2011  Spring   0   1.0   0      0.0  Saturday           0
1        2  1/1/2011  Spring   0   1.0   1      0.0  Saturday           0
2        3  1/1/2011  Spring   0   1.0   2      0.0  Saturday           0
3        4  1/1/2011  Spring   0   1.0   3      0.0  Saturday           0
4        5  1/1/2011  Spring   0   1.0   4      0.0  Saturday           0

   weathersit  temp   atemp   hum  windspeed  casual  registered   cnt
0         1.0  0.24  0.2879  0.81        0.0     3.0        13.0  16.0
1         1.0  0.22  0.2727  0.80        0.0     8.0        32.0  40.0
2         1.0  0.22  0.2727  0.80        0.0     5.0        27.0  32.0
3         1.0  0.24  0.2879  0.75        0.0     3.0        10.0  13.0
4         1.0  0.24  0.2879  0.75        0.0     0.0         1.0   1.0
-------------------------------------------------------
```

Figure 4 df.head() output

As for the next function used, df.info() showed us how many non-Null values in each column, also providing the type of each column (int, float, object), object values need to then be encoded into integers so we can insert them in the machine learning.

```
df.info()

<class 'pandas.core.frame.DataFrame'>
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
dtypes: float64(10), int64(4), object(3)
memory usage: 2.3+ MB
None
-----------------------------------------------------
```

Figure 5 df.info() Output

The df.describe() provides statistical insights across all numerical columns:

- **Central Tendency and Dispersion**: The mean, standard deviation, and interquartile ranges for continuous variables like temp, atemp, hum, and windspeed give an idea about the central tendencies and variability. For instance, temp and atemp have similar distributions, which is expected as they are both measures of perceived temperature.
- **Data Integrity Issues**: Some max values appear erroneous (e.g., temp max at 7 and atemp max at 14), indicating potential data errors or outliers that need to be addressed.
- **User Data**: The casual, registered, and cnt fields describe usage statistics, with cnt being the aggregate of casual and registered, providing a basis for deep dives into user behavior and rental trends.

```
df.describe()

          instant           yr          mnth            hr       holiday  \
count  17379.0000  17379.000000  17378.000000  17379.000000  17367.000000
mean    8690.0000      0.502561      6.538094     11.546752      0.028790
std     5017.0295      0.500008      3.438618      6.914405      0.167221
min        1.0000      0.000000      1.000000      0.000000      0.000000
25%     4345.5000      0.000000      4.000000      6.000000      0.000000
50%     8690.0000      1.000000      7.000000     12.000000      0.000000
75%    13034.5000      1.000000     10.000000     18.000000      0.000000
max    17379.0000      1.000000     12.000000     23.000000      1.000000

         workingday    weathersit          temp         atemp           hum  \
count  17379.000000  17376.000000  16930.000000  17324.000000  17086.000000
mean       0.682721      1.425184      0.496731      0.480431      0.626541
std        0.465431      0.639367      0.211812      0.289931      0.192888
min        0.000000      1.000000      0.020000      0.000000      0.000000
25%        0.000000      1.000000      0.340000      0.333300      0.470000
50%        1.000000      1.000000      0.500000      0.484800      0.630000
75%        1.000000      2.000000      0.660000      0.621200      0.780000
max        1.000000      4.000000      7.000000     14.000000      1.000000

          windspeed        casual    registered           cnt
count  17071.000000  17344.000000  17363.000000  17367.000000
mean       0.192688     35.642412    153.883603    189.520182
std        0.242883     49.261964    151.387243    181.402041
min        0.000000      0.000000      0.000000      1.000000
25%        0.104500      4.000000     34.000000     40.000000
50%        0.194000     17.000000    116.000000    142.000000
75%        0.253700     48.000000    220.000000    281.000000
max       17.000000    367.000000    886.000000    977.000000
----------------------------------------------------------
```

Figure 6 df,descripe() output

the count of missing values in each column, which we will handle afterwards

```
df.isnull().sum()
```

	0
instant	0
dteday	0
season	0
yr	0
mnth	1
hr	0
holiday	12
weekday	1
workingday	0
weathersit	3
temp	449
atemp	55
hum	293
windspeed	308
casual	35
registered	16
cnt	12

dtype: int64

Figure 7 df.isnull().sum() ouput

Proposed Methods for Handling Missing Data

1. **Simple Imputation**:
   - For columns with few missing values (mnth, holiday, weekday, weathersit, registered, cnt), fill missing entries with the most frequent value (mode) or median, depending on the data type and distribution.
2. **Predictive Imputation**:
   - For casual, which has a moderate number of missing values, consider using other correlated variables in the dataset to predict missing entries, potentially through a simple regression model or a machine learning algorithm like k-Nearest Neighbors.
3. **Advanced Techniques for Extensive Missing Data**:
   - For temp, atemp, hum, and windspeed, explore methods like multiple imputation, which accounts for uncertainty in the imputation process, or use time-series specific techniques like forward fill, backward fill, or linear interpolation, particularly since the data is sequential and time-stamped.

Afterwards, we did some visualization to get some better understanding:

```python
# Countplot for registered
sns.countplot(x='mnth', data=df)
plt.title('Count of mnth')
plt.show()
```

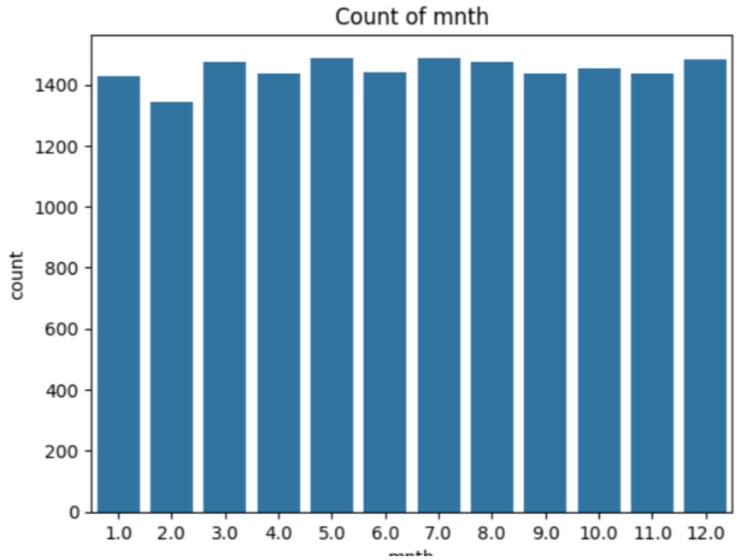


Figure 8 Count of mnth ploting

```python
43] # Countplot for survival based on class
sns.countplot(x='mnth', hue='workingday', data=df)
plt.title('workingday Count based on mnth')
plt.show()
```

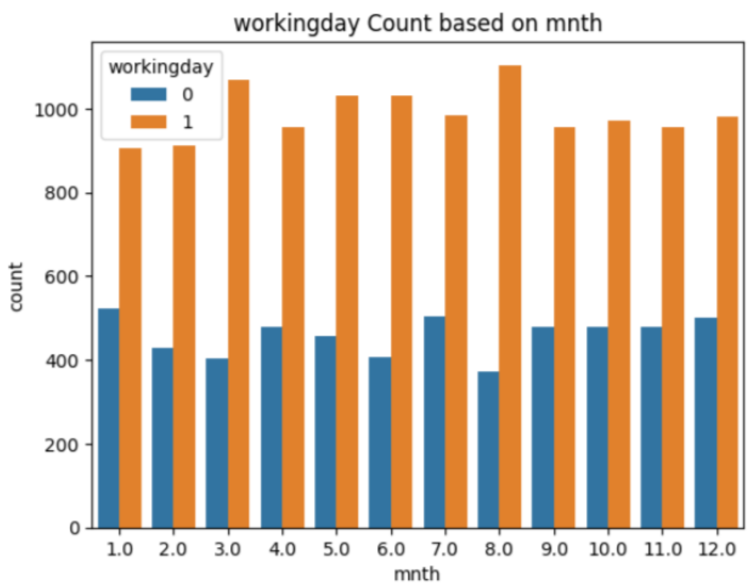


Figure 9 WorkingDay Count based on mnth

```
# Boxplot for fare distribution based on class
sns.boxplot(x='hr', y='season', data=df)
plt.title('Fare hr based on season')
plt.show()
```

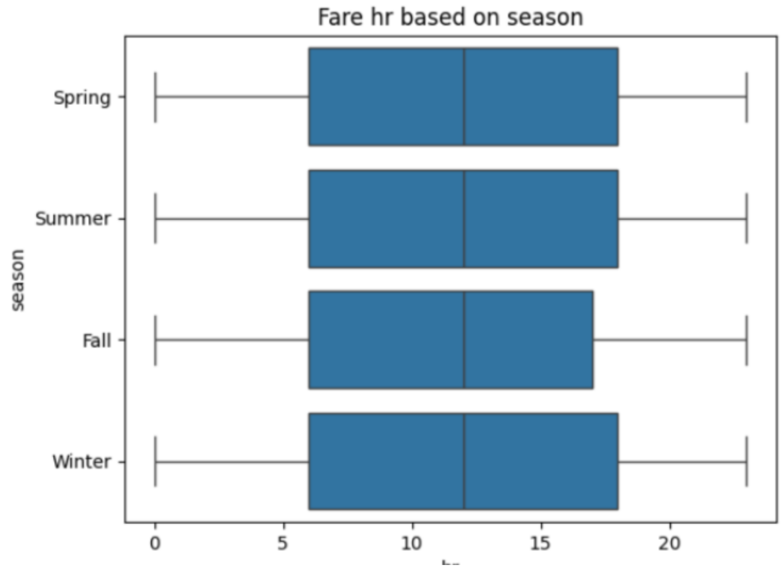


Figure 10 Boxplot for fare distribution based on season

```python
# Heatmap for correlation
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='coolwarm', linewidths=5.0)
plt.title('Correlation Heatmap')
plt.show()
```

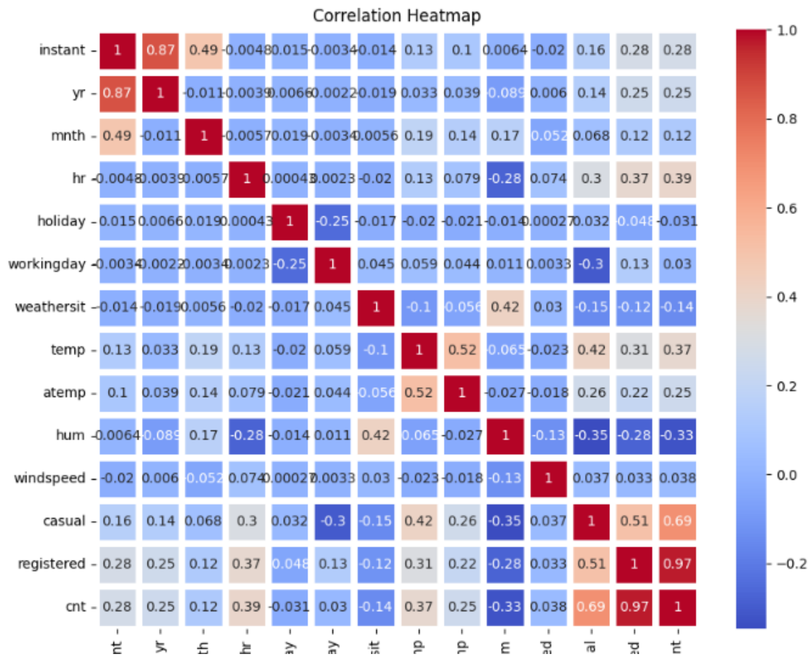


Figure 11 Heatmap for correlation

2.3 Handling Missing Data

As we saw from the output of print(df.isnull().sum()), there was some columns that has some null values. We choose some operations to do so we can deal with them.

1- Simple Imputation For columns with few missing values, we'll fill in missing entries with the mode (most common value).

2- Predictive Imputation For casual, which has a moderate number of missing values, we'll use a simple model like k-Nearest Neighbors for imputation. Here, we'll use IterativeImputer from scikit-learn, which models each feature with missing values as a function of other features in a round-robin fashion

3- Advanced Techniques for Extensive Missing Data For columns like temp, atemp, hum, and windspeed with substantial missing values, we will use linear interpolation, which is suitable for time series data.

Simple Imputation For columns with few missing values, we'll fill in missing entries with the mode (most common value).

```
[19] # Simple Imputation with the mode for columns with few missing values
     mode_values = df[['mnth', 'holiday', 'weekday', 'weathersit', 'registered', 'cnt']].mode().iloc[0]
     df[['mnth', 'holiday', 'weekday', 'weathersit', 'registered', 'cnt']] = df[['mnth', 'holiday', 'weekday', 'weathersit', 'registered', 'cnt']].fillna(mode_values)
```

Predictive Imputation For casual, which has a moderate number of missing values, we'll use a simple model like k-Nearest Neighbors for imputation. Here, we'll use IterativeImputer from scikit-learn, which models each feature with missing values as a function of other features in a round-robin fashion

```
[20] from sklearn.experimental import enable_iterative_imputer
     from sklearn.impute import IterativeImputer
     from sklearn.ensemble import RandomForestRegressor

     # Predictive imputation using IterativeImputer with RandomForestRegressor
     imp = IterativeImputer(estimator=RandomForestRegressor(), initial_strategy='mean', max_iter=10, random_state=0)
     df[['casual']] = imp.fit_transform(df[['casual']])
```

Advanced Techniques for Extensive Missing Data For columns like temp, atemp, hum, and windspeed with substantial missing values, we will use linear interpolation, which is suitable for time series data.

```
[21] # Advanced imputation using linear interpolation for time series data
     df['temp'] = df['temp'].interpolate(method='linear')
     df['atemp'] = df['atemp'].interpolate(method='linear')
     df['hum'] = df['hum'].interpolate(method='linear')
     df['windspeed'] = df['windspeed'].interpolate(method='linear')
```

Figure 12 Dealing with missing values

```
[22] # Validate the imputation
     print(df.isnull().sum())

instant       0
dteday        0
season        0
yr            0
mnth          0
hr            0
holiday       0
weekday       0
workingday    0
weathersit    0
temp          0
atemp         0
hum           0
windspeed     0
casual        0
registered    0
cnt           0
dtype: int64
```

Figure 13 Null values after handling them

2.4 Encoding into numerical values

We have done the following for this purpose:

1- Using Scikit-Learn for Label Encoding for "season" and "weekday"

2- Encoding the dteday column the dteday column represents dates and should be transformed into a more useful numerical format: This conversion separates the date into distinct year, month, and day components, which are more useful for regression analysis and other modeling techniques than a string format.

Encoding into numerical data

Using Scikit-Learn for Label Encoding for "season" and "weekday"

```
[23] from sklearn.preprocessing import LabelEncoder

     # Initialize label encoder
     label_encoder = LabelEncoder()

     # Apply label encoder on 'season' and 'weekday'
     df['season'] = label_encoder.fit_transform(df['season'])
     df['weekday'] = label_encoder.fit_transform(df['weekday'])
```

Encoding the dteday column The dteday column represents dates and should be transformed into a more useful numerical format: This conversion separates the date into distinct year, month, and day components, which are more useful for regression analysis and other modeling techniques than a string format.

```
[24] # Convert 'dteday' to datetime type
     df['dteday'] = pd.to_datetime(df['dteday'])

     # Extract year, month, and day as separate columns
     df['year'] = df['dteday'].dt.year
     df['month'] = df['dteday'].dt.month
     df['day'] = df['dteday'].dt.day

     # Optionally, drop 'dteday' if no longer needed
     df.drop('dteday', axis=1, inplace=True)
```

Figure 14 Code used for Encoding

Result of encoding can be seen in the figure below

```
[25] print(df.head())
     print(df.info())
```

```
   instant  season  yr  mnth  hr  holiday  weekday  workingday  weathersit  \
0        1       1   0   1.0   0      0.0        2           0         1.0
1        2       1   0   1.0   1      0.0        2           0         1.0
2        3       1   0   1.0   2      0.0        2           0         1.0
3        4       1   0   1.0   3      0.0        2           0         1.0
4        5       1   0   1.0   4      0.0        2           0         1.0

   temp   atemp   hum  windspeed  casual  registered   cnt  year  month  day
0  0.24  0.2879  0.81        0.0     3.0        13.0  16.0  2011      1    1
1  0.22  0.2727  0.80        0.0     8.0        32.0  40.0  2011      1    1
2  0.22  0.2727  0.80        0.0     5.0        27.0  32.0  2011      1    1
3  0.24  0.2879  0.75        0.0     3.0        10.0  13.0  2011      1    1
4  0.24  0.2879  0.75        0.0     0.0         1.0   1.0  2011      1    1
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17379 entries, 0 to 17378
Data columns (total 19 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instant     17379 non-null  int64
 1   season      17379 non-null  int64
 2   yr          17379 non-null  int64
 3   mnth        17379 non-null  float64
 4   hr          17379 non-null  int64
 5   holiday     17379 non-null  float64
 6   weekday     17379 non-null  int64
 7   workingday  17379 non-null  int64
 8   weathersit  17379 non-null  float64
 9   temp        17379 non-null  float64
 10  atemp       17379 non-null  float64
 11  hum         17379 non-null  float64
 12  windspeed   17379 non-null  float64
 13  casual      17379 non-null  float64
 14  registered  17379 non-null  float64
 15  cnt         17379 non-null  float64
 16  year        17379 non-null  int32
 17  month       17379 non-null  int32
 18  day         17379 non-null  int32
dtypes: float64(10), int32(3), int64(6)
```

Figure 15 Data after encoding

Difference between original data, and data after scaling

```
Standard Scaled Data:
   instant  season  yr  mnth  hr  holiday  weekday  workingday  weathersit  \
0        1       1   0   1.0   0      0.0        2           0         1.0
1        2       1   0   1.0   1      0.0        2           0         1.0
2        3       1   0   1.0   2      0.0        2           0         1.0
3        4       1   0   1.0   3      0.0        2           0         1.0
4        5       1   0   1.0   4      0.0        2           0         1.0

       temp     atemp       hum  windspeed    casual  registered   cnt  year  \
0 -1.227782 -0.665588  0.952034  -0.801944 -0.663317   -0.929741  16.0  2011
1 -1.322727 -0.718060  0.900048  -0.801944 -0.561713   -0.804231  40.0  2011
2 -1.322727 -0.718060  0.900048  -0.801944 -0.622675   -0.837260  32.0  2011
3 -1.227782 -0.665588  0.640119  -0.801944 -0.663317   -0.949559  13.0  2011
4 -1.227782 -0.665588  0.640119  -0.801944 -0.724279   -1.009011   1.0  2011

   month  day
0      1    1
1      1    1
2      1    1
3      1    1
4      1    1
-----------------------------
Original Data:
   instant  season  yr  mnth  hr  holiday  weekday  workingday  weathersit  \
0        1       1   0   1.0   0      0.0        2           0         1.0
1        2       1   0   1.0   1      0.0        2           0         1.0
2        3       1   0   1.0   2      0.0        2           0         1.0
3        4       1   0   1.0   3      0.0        2           0         1.0
4        5       1   0   1.0   4      0.0        2           0         1.0

   temp   atemp   hum  windspeed  casual  registered   cnt  year  month  day
0  0.24  0.2879  0.81        0.0     3.0        13.0  16.0  2011      1    1
1  0.22  0.2727  0.80        0.0     8.0        32.0  40.0  2011      1    1
2  0.22  0.2727  0.80        0.0     5.0        27.0  32.0  2011      1    1
3  0.24  0.2879  0.75        0.0     3.0        10.0  13.0  2011      1    1
4  0.24  0.2879  0.75        0.0     0.0         1.0   1.0  2011      1    1
```

2.5 Identifying Outliers

As for this part exactly, we have tried many operations identifying the outliers, as which they were:

1- Z-score
2- IQR detection
3- Sns Boxplots

```
[26] from scipy.stats import zscore

     # Calculate Z-scores of the data
     df_numeric = df.select_dtypes(include=[np.number])  # selecting numeric columns
     df_numeric['z_score_temp'] = zscore(df_numeric['temp'])

     # Filter entries that have a temperature z-score greater than 3 or less than -3
     outliers_temp = df_numeric[(df_numeric['z_score_temp'] > 3) | (df_numeric['z_score_temp'] < -3)]
     print("Temperature Outliers based on Z-score:")
     print(outliers_temp)
```

```
Temperature Outliers based on Z-score:
       instant  season  yr  mnth  hr  holiday  weekday  workingday  \
15531    15532       3   1  10.0   8      0.0        3           0
16534    16535       3   1  11.0  16      0.0        1           1
16535    16536       3   1  11.0  17      0.0        1           1
16536    16537       3   1  11.0  18      0.0        1           1
16537    16538       3   1  11.0  19      0.0        1           1
16538    16539       3   1  11.0  20      0.0        1           1
16539    16540       3   1  11.0  21      0.0        1           1
16540    16541       3   1  11.0  22      0.0        1           1

       weathersit  temp   atemp   hum  windspeed  casual  registered    cnt  \
15531         1.0   1.6  0.4394  0.77    0.30084    28.0       104.0  132.0
16534         1.0   2.0  0.4394  0.30    0.00000    49.0       297.0  346.0
16535         1.0   3.0  0.4242  0.32    0.00000    13.0       540.0  553.0
16536         1.0   4.0  0.3485  0.50    0.16420    19.0       502.0  521.0
16537         1.0   4.0  0.3636  0.53    0.00000    16.0       355.0  371.0
16538         1.0   5.0  0.3636  0.49    0.00000    12.0       265.0  277.0
16539         1.0   6.0  0.3636  0.49    0.00000     9.0       172.0  181.0
16540         1.0   7.0  0.3485  0.61    0.00000     3.0       101.0  104.0

       year  month  day  z_score_temp
15531  2012     10   14      5.228441
16534  2012     11   26      7.127331
16535  2012     11   26     11.874554
16536  2012     11   26     16.621777
16537  2012     11   26     16.621777
16538  2012     11   26     21.369001
16539  2012     11   26     26.116224
16540  2012     11   26     30.863447
```

Figure 16 Temperature Outliers based on Z-score

Using IQR to Detect Outliers in Humidity

```
[27] # Calculate Q1, Q3, and IQR
     Q1 = df['hum'].quantile(0.25)
     Q3 = df['hum'].quantile(0.75)
     IQR = Q3 - Q1

     # Define outliers as those values outside the IQR * 1.5 criterion
     outliers_hum = df[(df['hum'] < (Q1 - 1.5 * IQR)) | (df['hum'] > (Q3 + 1.5 * IQR))]
     print("Humidity Outliers based on IQR:")
     print(outliers_hum)
```

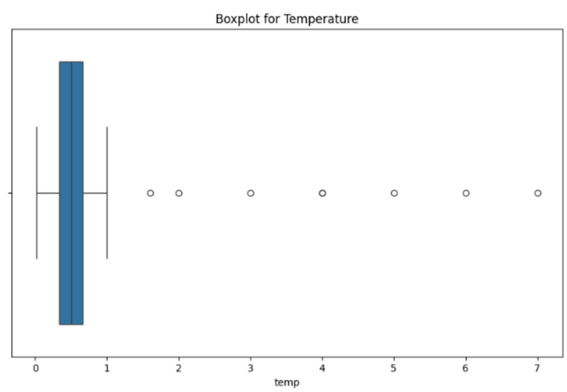
```
  55    1556    1  0  3.0   6   0.0   4   1   3.0
  56    1557    1  0  3.0   7   0.0   4   1   3.0
1557    1558    1  0  3.0   8   0.0   4   1   3.0
1558    1559    1  0  3.0   9   0.0   4   1   3.0
1559    1560    1  0  3.0  10   0.0   4   1   3.0
1560    1561    1  0  3.0  11   0.0   4   1   3.0
1561    1562    1  0  3.0  12   0.0   4   1   3.0
1562    1563    1  0  3.0  13   0.0   4   1   3.0
1563    1564    1  0  3.0  14   0.0   4   1   3.0
1564    1565    1  0  3.0  15   0.0   4   1   3.0
1565    1566    1  0  3.0  16   0.0   4   1   3.0
1566    1567    1  0  3.0  17   0.0   4   1   2.0
1567    1568    1  0  3.0  18   0.0   4   1   3.0
1568    1569    1  0  3.0  19   0.0   4   1   3.0
1569    1570    1  0  3.0  20   0.0   4   1   3.0
1570    1571    1  0  3.0  21   0.0   4   1   3.0
1571    1572    1  0  3.0  22   0.0   4   1   2.0
1572    1573    1  0  3.0  23   0.0   4   1   3.0
```

Figure 17 Using IQR to Detect Outliers in Humidity

```
[30] import matplotlib.pyplot as plt
     import seaborn as sns

     # Boxplot for temperature
     plt.figure(figsize=(10, 6))
     sns.boxplot(x=df['temp'])
     plt.title('Boxplot for Temperature')
     plt.show()
```

Figure 19 Boxplot code



Boxplot for Atemperature

Figure 18 Boxplot output

2.6 Identifying Outliers

In this part, I have used StandardScaler on these features: 'temp', 'atemp', 'hum', 'windspeed', 'casual', 'registered', As for the rest, I chose to keep them as they are because they seem not needing any scaling.

```
from sklearn.preprocessing import StandardScaler

standard_scaler = StandardScaler()

# I have decided to scale only these values since other values don't really need scaling
# List of columns to scale, assuming you want to scale all numeric data
columns_to_scale = ['temp', 'atemp', 'hum', 'windspeed', 'casual', 'registered']

# Apply Standard Scaling
df[columns_to_scale] = standard_scaler.fit_transform(df[columns_to_scale])

# Print the first few rows to check the result of scaling
print("\nStandard Scaled Data:")
print(df.head())
```

```
Standard Scaled Data:
   instant  season  yr  mnth  hr  holiday  weekday  workingday  weathersit  \
0        1       1   0   1.0   0      0.0        2           0         1.0
1        2       1   0   1.0   1      0.0        2           0         1.0
2        3       1   0   1.0   2      0.0        2           0         1.0
3        4       1   0   1.0   3      0.0        2           0         1.0
4        5       1   0   1.0   4      0.0        2           0         1.0

       temp     atemp       hum  windspeed    casual  registered   cnt  year  \
0 -1.227782 -0.665588  0.952034  -0.801944 -0.663317   -0.929741  16.0  2011
1 -1.322727 -0.718060  0.900048  -0.801944 -0.561713   -0.804231  40.0  2011
2 -1.322727 -0.718060  0.900048  -0.801944 -0.622675   -0.837260  32.0  2011
3 -1.227782 -0.665588  0.640119  -0.801944 -0.663317   -0.949559  13.0  2011
4 -1.227782 -0.665588  0.640119  -0.801944 -0.724279   -1.009011   1.0  2011

   month  day
0      1    1
1      1    1
2      1    1
3      1    1
4      1    1
```

Figure 20 Scaling

2.7 Feature Selection

This part had been done using Random Forest for Feature Importance. We have dropped the target “cnt” and started calculating the importance of the features to the target.

```
df_standard_scaled = df.copy()

# Assuming 'df_standard_scaled' is the DataFrame you want to use
X = df_standard_scaled.drop('cnt', axis=1)  # Features
y = df_standard_scaled['cnt']               # Target variable

# Initialize the model
rf = RandomForestRegressor(n_estimators=100, random_state=42)

# Fit the model
rf.fit(X, y)

# Get feature importances
importances = rf.feature_importances_
feature_names = X.columns

# Create a DataFrame to view the features and their importance scores
feature_importances = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
feature_importances = feature_importances.sort_values(by='Importance', ascending=False)
print(feature_importances)
```

```
       Feature  Importance
14  registered    0.947185
13      casual    0.051966
0      instant    0.000158
11         hum    0.000125
4           hr    0.000102
7   workingday    0.000094
12   windspeed    0.000064
8   weathersit    0.000063
17         day    0.000047
6      weekday    0.000045
10       atemp    0.000043
9         temp    0.000040
16       month    0.000024
1       season    0.000021
3         mnth    0.000014
2           yr    0.000005
15        year    0.000002
5      holiday    0.000002
```

Figure 21 feature selection-1

Since both registered and casual users are directly related to the total rental counts (cnt), It also suggests that the model is primarily using these two features to make predictions, which is intuitive but might not be particularly useful if our goal is to predict future counts without knowing these breakdowns ahead of time. So, we decided to drop them off also.

while the goal is to predict total bike rentals without prior knowledge of user type breakdowns (casual vs. registered), I am rebuilding the model without the casual and registered features. This would provide insights into what other factors influence rental counts and how much they matter.

```
[42] # Drop 'casual' and 'registered' features
     X_revised = df_standard_scaled.drop(['cnt', 'casual', 'registered'], axis=1)
     y_revised = df_standard_scaled['cnt']

     # Fit the model on revised data
     rf_revised = RandomForestRegressor(n_estimators=100, random_state=42)
     rf_revised.fit(X_revised, y_revised)

     # Get revised feature importances
     revised_importances = rf_revised.feature_importances_
     revised_feature_names = X_revised.columns
     revised_feature_importances = pd.DataFrame({'Feature': revised_feature_names, 'Importance': revised_importances})
     revised_feature_importances = revised_feature_importances.sort_values(by='Importance', ascending=False)
     print(revised_feature_importances)
```

```
        Feature  Importance
4            hr    0.583847
0       instant    0.165432
7    workingday    0.084162
9          temp    0.068057
10        atemp    0.024966
11          hum    0.022167
8    weathersit    0.015734
6       weekday    0.011088
15          day    0.010654
12    windspeed    0.006640
14        month    0.002021
3          mnth    0.002017
1        season    0.001609
5       holiday    0.001474
2            yr    0.000075
13         year    0.000057
```

Figure 22 feature selection-2

explaining the above:

Revised Feature Importance Analysis

1- Hour of Day (hr): The most significant feature with an importance of approximately 58.4%. This indicates that the time of day is crucial for predicting bike rentals, which aligns with daily patterns of human activity (e.g., commuting times in the morning and evening).

2- Instant (instant): Surprisingly, this feature, which likely represents a unique identifier for each record, holds substantial importance at 16.5%. This might suggest some chronological trends in the data or could be an artifact of how data was collected or indexed.

3- Working Day (workingday): With an importance of 8.4%, this feature signifies whether a day is a regular working day or not, influencing rental patterns due to commuting behavior.

4- Temperature (temp): Contributing 6.8% importance, this reflects the intuitive understanding that weather conditions affect outdoor activities like biking.

5- Feels Like Temperature (atemp) and Humidity (hum): These also play roles but to a lesser extent, emphasizing the effect of perceived environmental conditions on rental decisions.

6- Weather Situation (weathersit) and Windspeed (windspeed): Lesser but notable effects, indicating adverse weather can deter bike usage.

Interpretation:

1- Time Dependency: The high importance of hr and instant underscores the time-sensitive nature of bike rentals. These features help capture patterns across different times of the day and potentially across the dataset's timeline.

2- Weather and Environmental Factors: temp, atemp, hum, and weathersit confirm the expected influence of weather on biking habits. Even though their individual importances are not as high as hr, they collectively account for a significant portion of the predictive power.

3- Work-Related Usage: The significance of workingday aligns with usage patterns where bikes are likely used more on workdays, possibly for commuting to work or other regular activities.

2.8 Dimensionality Reduction and Modeling

RandomForestRegressor was used due to target being continuous. We have tried training the model before using PCA to see the difference before and after adding it.

Machine Learning

Before Applying PCA

```
[50] # Assuming df_standard_scaled is your DataFrame after standard scaling
features = df_standard_scaled.drop(['cnt', 'casual', 'registered'], axis=1)
target = df_standard_scaled['cnt']

# Split data into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)

# Train a Random Forest model
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
from sklearn.metrics import mean_squared_error, r2_score
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

print(f"Mean Squared Error: {mse}")
print(f"R-squared: {r2}")
```

```
Mean Squared Error: 1687.650786306099
R-squared: 0.9467103835046389
```

Figure 23 Training before PCA

Then we applied the PCA to the model, choosing to keep 95% of the variance.

Applying PCA

```
[48] from sklearn.decomposition import PCA
     from sklearn.ensemble import RandomForestRegressor
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import mean_squared_error, r2_score

     # Assuming 'df_standard_scaled' is pre-processed and scaled
     X = df_standard_scaled.drop(['cnt', 'casual', 'registered'], axis=1)
     y = df_standard_scaled['cnt']

     # Initialize PCA: Choosing to keep 95% of the variance
     pca = PCA(n_components=0.95)
     X_pca = pca.fit_transform(X)

     # Split data into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)

     # Train a Random Forest model
     rf = RandomForestRegressor(n_estimators=100, random_state=42)
     rf.fit(X_train, y_train)

     # Predict and evaluate the model
     predictions = rf.predict(X_test)
     mse = mean_squared_error(y_test, predictions)
     r2 = r2_score(y_test, predictions)

     print(f"Mean Squared Error: {mse}")
     print(f"R-squared: {r2}")
     print(f"Number of components: {pca.n_components_}")
```

```
Mean Squared Error: 7248.354312945915
R-squared: 0.7711244383651028
Number of components: 1
```

Figure 24 training with PCA at 0.95 variance

Then a problem with lower R-squared was noticed as we realized that it needs changing, so we had to experiment with adjusting the number of principal components, instead of setting a variance ratio, try using a fixed number of components and evaluate how the model performance changes.

```
# Example: Trying different numbers of components
for n in range(1, X.shape[1] + 1):
    pca = PCA(n_components=n)
    X_pca = pca.fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)
    rf.fit(X_train, y_train)
    print(f"n_components={n}, R-squared={rf.score(X_test, y_test)}")
```

```
n_components=1, R-squared=0.7711244383651028
n_components=2, R-squared=0.7778542858409898
n_components=3, R-squared=0.7968027242017669
n_components=4, R-squared=0.7928452929798234
n_components=5, R-squared=0.9067033306152941
n_components=6, R-squared=0.9122362470589157
n_components=7, R-squared=0.9144642729688104
n_components=8, R-squared=0.9208491758302232
n_components=9, R-squared=0.9223169763114935
n_components=10, R-squared=0.9229657307795179
n_components=11, R-squared=0.9246032452138943
n_components=12, R-squared=0.9397241308887125
n_components=13, R-squared=0.9402479384670723
n_components=14, R-squared=0.9405685421708772
n_components=15, R-squared=0.9405983072077418
n_components=16, R-squared=0.9404233451921579
```

Figure 25 trying different number of components

After the seen output in figure 25, we figured out we should use number of components equal to 14.

```
[49] from sklearn.decomposition import PCA
     from sklearn.ensemble import RandomForestRegressor
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import mean_squared_error, r2_score

     # Assuming 'df_standard_scaled' is pre-processed and scaled
     X = df_standard_scaled.drop(['cnt', 'casual', 'registered'], axis=1)
     y = df_standard_scaled['cnt']

     # Initialize PCA: adjusting the number of principal components. Instead of setting a variance ratio (14)
     pca = PCA(n_components=14)
     X_pca = pca.fit_transform(X)

     # Split data into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)

     # Train a Random Forest model
     rf = RandomForestRegressor(n_estimators=100, random_state=42)
     rf.fit(X_train, y_train)

     # Predict and evaluate the model
     predictions = rf.predict(X_test)
     mse = mean_squared_error(y_test, predictions)
     r2 = r2_score(y_test, predictions)

     print(f"Mean Squared Error: {mse}")
     print(f"R-squared: {r2}")
     print(f"Number of components: {pca.n_components_}")
```

```
Mean Squared Error: 1882.1592860471806
R-squared: 0.9405685421708772
Number of components: 14
```

Figure 26 training with n_components = 14

And got the final values which were:

1- Mean Squared Error: 1882.1592860471806
2- R-squared: 0.9405685421708772

2.9 Comparison

As for the comparison between the raw data and the preprocessed data, we chose the raw data which were only encoded and compared the results with the preprocesses data.

model training on the raw data (before feature filtering, transformation, and reduction)

```
[52]
rawdf = pd.read_csv('/content/hours.csv')


# Initialize label encoder
label_encoder = LabelEncoder()

# Apply label encoder on 'season' and 'weekday'
rawdf['season'] = label_encoder.fit_transform(rawdf['season'])
rawdf['weekday'] = label_encoder.fit_transform(rawdf['weekday'])

# Convert 'dteday' to datetime type
rawdf['dteday'] = pd.to_datetime(rawdf['dteday'])

# Extract year, month, and day as separate columns
rawdf['year'] = rawdf['dteday'].dt.year
rawdf['month'] = rawdf['dteday'].dt.month
rawdf['day'] = rawdf['dteday'].dt.day

# Optionally, drop 'dteday' if no longer needed
rawdf.drop('dteday', axis=1, inplace=True)


# Assuming df_standard_scaled is your DataFrame after standard scaling
features = df_standard_scaled.drop(['cnt', 'casual', 'registered'], axis=1)
target = df_standard_scaled['cnt']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)

# Train a Random Forest model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

print(f"Mean Squared Error: {mse}")
print(f"R-squared: {r2}")
```

Figure 27 training of raw data

```
Mean Squared Error: 1687.650786306099
R-squared: 0.9467103835046389
```

Figure 28 raw data training output

We found that the preprocessing of the data really improved the model training.

3 Conclusion

The study concludes that while basic Decision Tree models offer a reasonable baseline for predicting bike rental counts, significant improvements can be achieved through hyperparameter tuning and feature engineering. The results emphasize the importance of incorporating multiple features and understanding their interactions to accurately predict demand in bike-sharing systems. Future work may explore more advanced models, such as Random Forests or Gradient Boosting Machines, to further enhance performance. The findings also suggest exploring more granular temporal features and considering ensemble methods or time series models for capturing complex interactions and dynamics within the data.