BIRZEIT UNIVERSITY

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

Case Study #2: Comparative Analysis of Classification Techniques: Random Forest (RF), Extreme Gradient Boosting (XGBoost), and Multilayer Perceptron (MLP)

Prepared by: Nafe Abubaker—1200047

Instructor: **Dr. Mohammad Jubran**

Assistant: **Eng. Hanan**

Date: **August,4,2024**

Abstract

This study presents a comparative analysis of three prominent machine learning classification techniques—Random Forest (RF), Extreme Gradient Boosting (XGBoost), and Multilayer Perceptron (MLP)—to predict bike rental demand in urban environments using the Capital Bikeshare dataset from 2011-2012. By categorizing demand into "low," "medium," "high," and "extreme" levels, the study evaluates the performance of each classifier based on accuracy, precision, recall, and F1-score. The dataset underwent extensive preprocessing, including handling missing data, encoding categorical variables, identifying outliers, and feature selection using Random Forest feature importance. The results demonstrate that XGBoost achieves the highest accuracy and balanced performance across all demand categories, outperforming RF and MLP. This analysis underscores the importance of model selection in predictive analytics for urban mobility, providing insights that can enhance bike-sharing service management and planning.

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 (From Case Study-1)

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

```python
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:

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

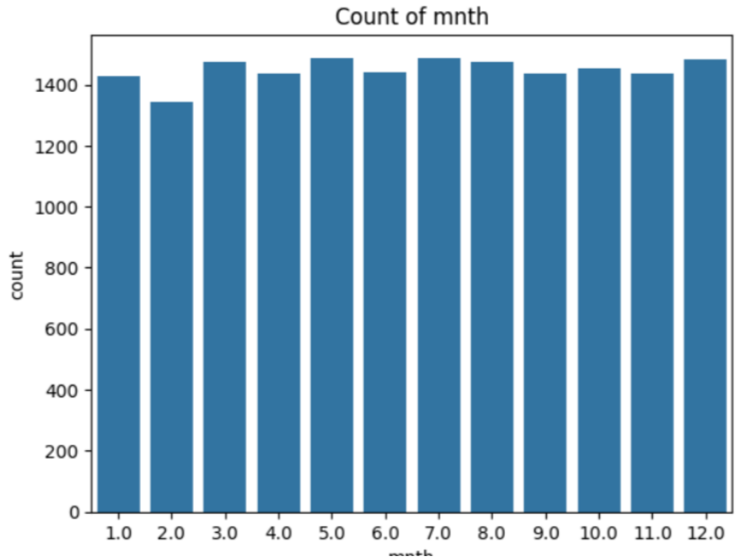

Figure 8 Count of mnth ploting

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

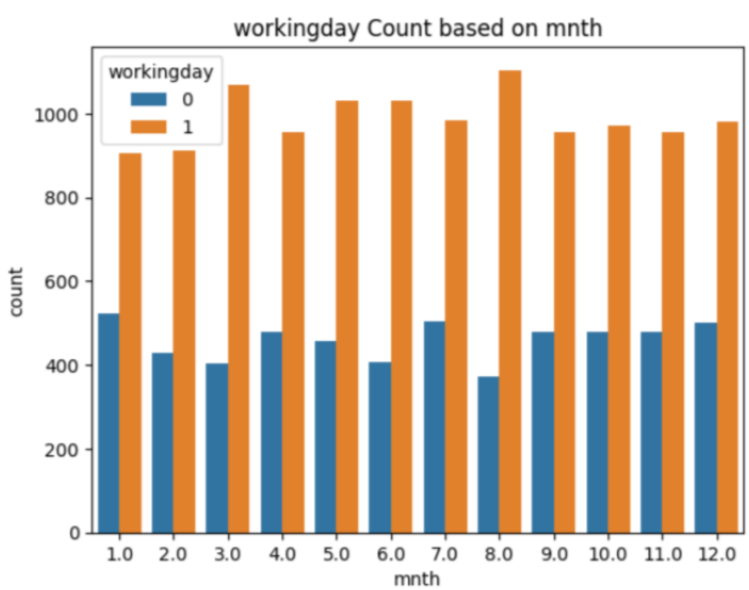

Figure 9 WorkingDay Count based on mnth

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

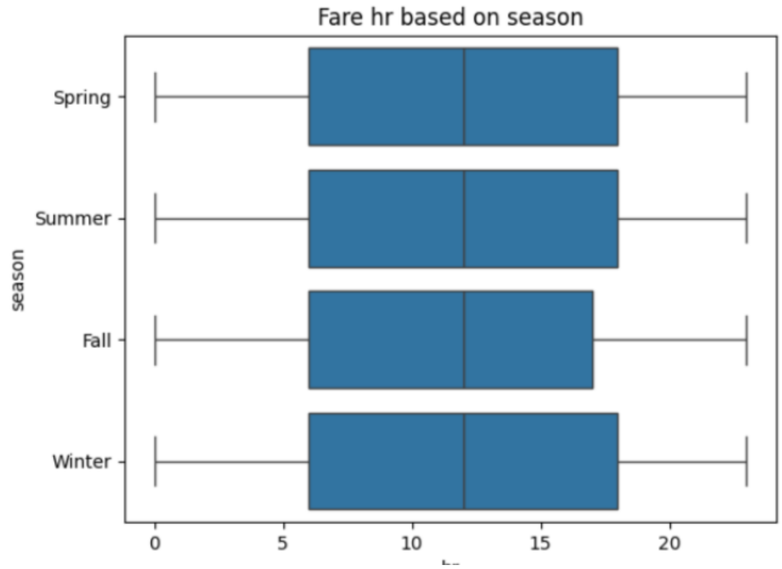

Figure 10 Boxplot for fare distribution based on season

```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()
```

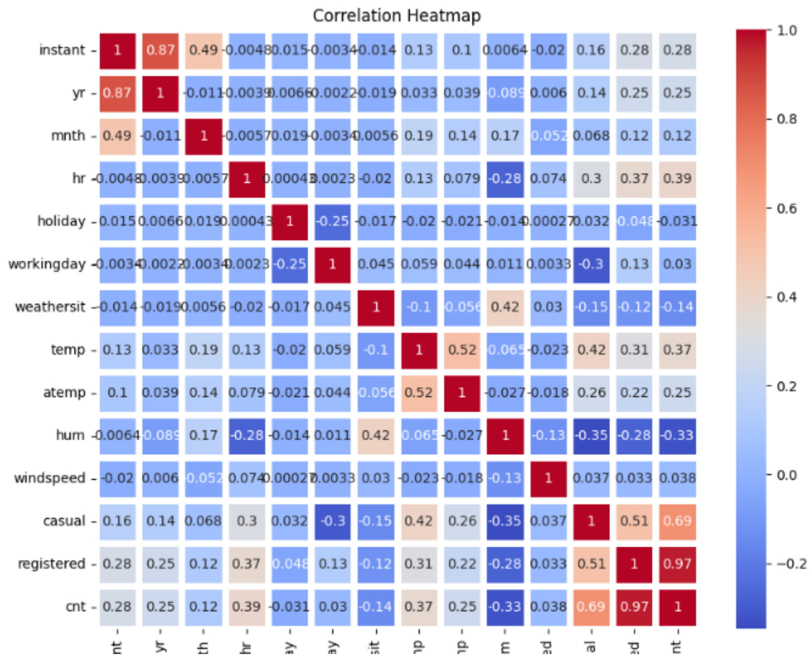

Figure 11 Heatmap for correlation

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

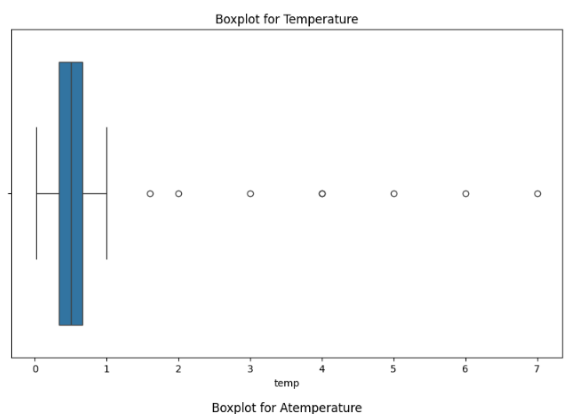

Figure 18 Boxplot output

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.

```python
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.

3 Comparative Analysis of Classification Techniques (Case study 2 starts from here)

For this case study, we took the already processed and cleaned data form the last case study, and moved on with it.

```
DataSet.head()
```

	instant	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt	year	month	day
0	1	1	0	1.0	0	0.0	2	0	1.0	-1.227782	-0.665588	0.952034	-0.801944	-0.663317	-0.929741	16.0	2011	1	1
1	2	1	0	1.0	1	0.0	2	0	1.0	-1.322727	-0.718060	0.900048	-0.801944	-0.561713	-0.804231	40.0	2011	1	1
2	3	1	0	1.0	2	0.0	2	0	1.0	-1.322727	-0.718060	0.900048	-0.801944	-0.622675	-0.837260	32.0	2011	1	1
3	4	1	0	1.0	3	0.0	2	0	1.0	-1.227782	-0.665588	0.640119	-0.801944	-0.663317	-0.949559	13.0	2011	1	1
4	5	1	0	1.0	4	0.0	2	0	1.0	-1.227782	-0.665588	0.640119	-0.801944	-0.724279	-1.009011	1.0	2011	1	1

Figure 23 dataset head values

3.1 Categorizing

We needed a function to categorize the count, so we defined the categories based on quartiles.

```python
def categorize_demand(cnt):
    if cnt <= DataSet['cnt'].quantile(0.25):
        return 'low'
    elif cnt <= DataSet['cnt'].quantile(0.50):
        return 'medium'
    elif cnt <= DataSet['cnt'].quantile(0.75):
        return 'high'
    else:
        return 'extreme'

# Apply the categorization function
DataSet['demand_category'] = DataSet['cnt'].apply(categorize_demand)
```

Figure 24 Categorizing

As we can see, the categorization is added (low, medium, high and extreme).

To see the results

```
DataSet.head()
```

	instant	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt	year	month	day	demand_category
0	1	1	0	1.0	0	0.0	2	0	1.0	-1.227782	-0.665588	0.952034	-0.801944	-0.663317	-0.929741	16.0	2011	1	1	low
1	2	1	0	1.0	1	0.0	2	0	1.0	-1.322727	-0.718060	0.900048	-0.801944	-0.561713	-0.804231	40.0	2011	1	1	low
2	3	1	0	1.0	2	0.0	2	0	1.0	-1.322727	-0.718060	0.900048	-0.801944	-0.622675	-0.837260	32.0	2011	1	1	low
3	4	1	0	1.0	3	0.0	2	0	1.0	-1.227782	-0.665588	0.640119	-0.801944	-0.663317	-0.949559	13.0	2011	1	1	low
4	5	1	0	1.0	4	0.0	2	0	1.0	-1.227782	-0.665588	0.640119	-0.801944	-0.724279	-1.009011	1.0	2011	1	1	low

Figure 25 Categorization added

3.2 Feature Selection

For Feature Selection and Data Splitting, we'll use the same set of features we considered earlier in case study 1, excluding casual and registered if they are still present to avoid data leakage.

```
[65] features = DataSet.drop(['cnt', 'casual', 'registered', 'demand_category'], axis=1)
     target = DataSet['demand_category']

     # Splitting the dataset 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)
```

Figure 26 Case2 Feature Selection

3.3 Model Training and Evaluation

We set up each model invdividually, then we will later call it

```
[66] from sklearn.ensemble import RandomForestClassifier
     from xgboost import XGBClassifier
     from sklearn.neural_network import MLPClassifier
     from sklearn.metrics import classification_report, accuracy_score

     # Random Forest
     rf = RandomForestClassifier(n_estimators=100, random_state=42)

     # XGBoost
     xgb = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42)

     # Multilayer Perceptron
     mlp = MLPClassifier(random_state=42)
```

Figure 27 Case2 Modeling

```python
# Initialize label encoder
label_encoder = LabelEncoder()

# Encode target labels
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

#-----------------------------------------------------------------------------

# Update the train_evaluate function to use encoded labels for XGBoost
def train_evaluate(model, X_train, y_train, X_test, y_test, is_xgb=False):
    if is_xgb:  # Check if the model is XGBoost
        # Fit and predict using encoded labels
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)
    else:
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)

    # Use inverse transform to get original labels for evaluation if it's XGBoost
    if is_xgb:
        predictions = label_encoder.inverse_transform(predictions)
        actual_y_test = label_encoder.inverse_transform(y_test)
    else:
        actual_y_test = y_test

    # Print accuracy and classification report
    print(f"Accuracy of {model.__class__.__name__}: {accuracy_score(actual_y_test, predictions)}")
    print(classification_report(actual_y_test, predictions))

    # Compute and print the confusion matrix
    cm = confusion_matrix(actual_y_test, predictions, labels=label_encoder.classes_)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
    plt.title(f'Confusion Matrix for {model.__class__.__name__}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()


#-----------------------------------------------------------------------------

# Train and evaluate Random Forest
train_evaluate(rf, X_train, y_train, X_test,y_test)

# Train and evaluate XGBoost
train_evaluate(xgb, X_train, y_train_encoded, X_test, y_test_encoded, is_xgb=True)

# Train and evaluate MLP
train_evaluate(mlp, X_train, y_train, X_test, y_test)
```

Figure 28 Model training and evaluating

3.4 Outputs

As for the outputs, we have printed the accuracy of each model, with their precision, recall, F1-score.

3.4.1 Random Forest Classifier

```
Accuracy of RandomForestClassifier: 0.80897583429229
              precision    recall  f1-score   support

     extreme       0.86      0.86      0.86       840
        high       0.71      0.73      0.72       858
         low       0.90      0.92      0.91       853
      medium       0.77      0.74      0.76       925

    accuracy                           0.81      3476
   macro avg       0.81      0.81      0.81      3476
weighted avg       0.81      0.81      0.81      3476
```

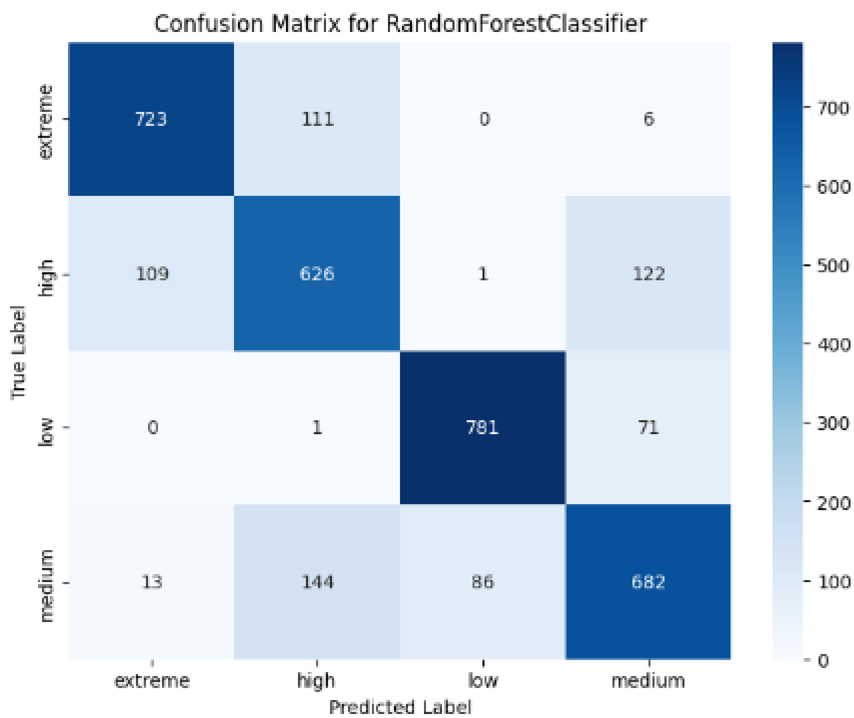

Figure 29 Random Forest Output

3.4.2 XGB Classifier

```
Accuracy of XGBClassifier: 0.8578826237054085
              precision    recall  f1-score   support

     extreme       0.88      0.89      0.89       840
        high       0.80      0.80      0.80       858
         low       0.93      0.91      0.92       853
      medium       0.83      0.83      0.83       925

    accuracy                           0.86      3476
   macro avg       0.86      0.86      0.86      3476
weighted avg       0.86      0.86      0.86      3476
```

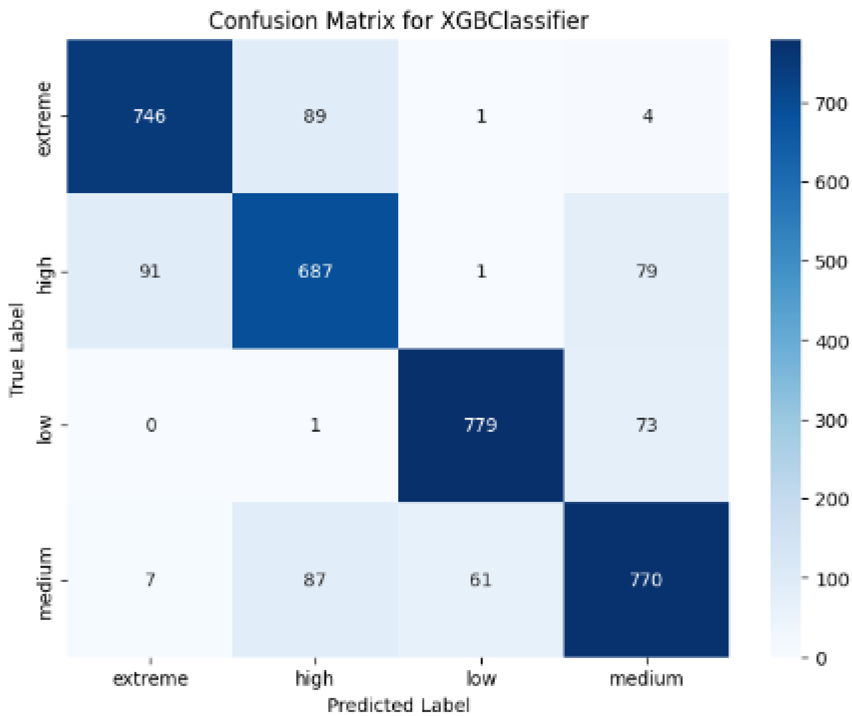

Figure 30 XGB output

3.4.3 MLP Classifier

```
Accuracy of MLPClassifier: 0.32163406214039125
              precision    recall  f1-score   support

     extreme       0.00      0.00      0.00       840
        high       0.29      0.91      0.44       858
         low       0.64      0.16      0.25       853
      medium       0.36      0.22      0.27       925

    accuracy                           0.32      3476
   macro avg       0.32      0.32      0.24      3476
weighted avg       0.32      0.32      0.24      3476

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1471: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1471: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1471: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

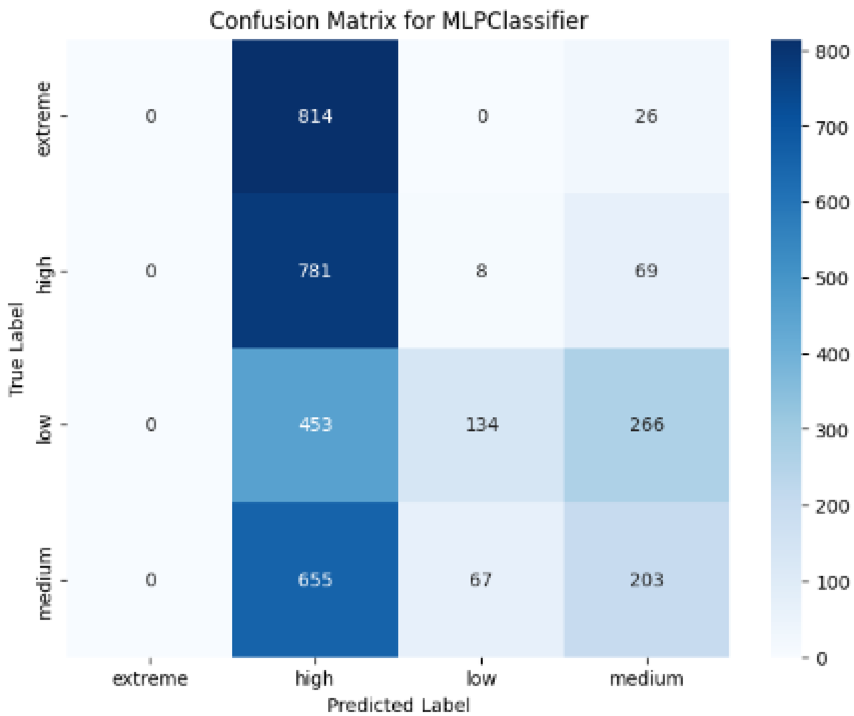

Figure 31 MLP Output

3.4.4 Plotting performance differences

```
import matplotlib.pyplot as plt
import numpy as np

# Define the models and their metrics
models = ['Random Forest', 'XGBoost', 'MLP']
accuracy = [0.80898, 0.85788, 0.32163]  # Actual accuracies from your results
precision = [0.81, 0.86, 0.32]  # Estimated from your provided average precision
recall = [0.81, 0.86, 0.32]  # Estimated from your provided average recall
f1 = [0.81, 0.86, 0.24]  # Estimated from your provided average F1-scores

x = np.arange(len(models))  # the label locations
width = 0.2  # the width of the bars

fig, ax = plt.subplots(figsize=(10, 6))

# Plotting accuracy
rects1 = ax.bar(x - width, accuracy, width, label='Accuracy', color='navy')

# Plotting precision
rects2 = ax.bar(x, precision, width, label='Precision', color='green')

# Plotting recall
rects3 = ax.bar(x + width, recall, width, label='Recall', color='red')

# Plotting F1-score
rects4 = ax.bar(x + 2*width, f1, width, label='F1-score', color='purple')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Comparison of Model Performance')
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.legend()

# Function to attach labels to bars
def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.2f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

# Adding labels to bars
autolabel(rects1)
autolabel(rects2)
autolabel(rects3)
autolabel(rects4)
```

Figure 32 Performance diffference code

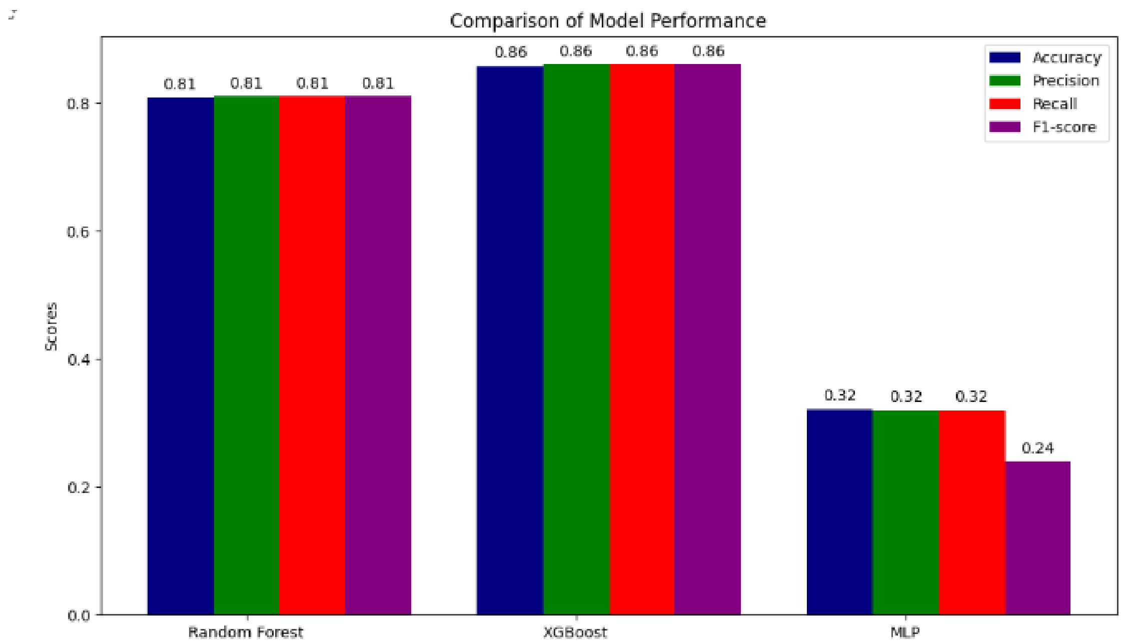

Figure 33 Performance difference plot output

4 Analyzation

4.1 Random Forest Classifier

- **Performance Metrics**: Accuracy of 80.90%
- **Confusion Matrix Analysis**:
    - Good classification performance across all categories.
    - Comparatively balanced precision, recall, and F1-scores, which indicates a well-rounded ability to classify all demand categories.
- **Strengths**:
    - Robust across a range of input variables and classifications.
    - Handles feature interactions naturally.
- **Weaknesses**:
    - May not capture complex nonlinear relationships as effectively as gradient boosting machines.
    - Can be biased towards features with more levels.

4.2 XGB Classifier

- **Performance Metrics**: Accuracy of 85.79%
- **Confusion Matrix Analysis**:
    - Very high accuracy and balanced scores across classes.
    - Exhibits slightly better precision and recall compared to RandomForest, particularly noticeable in extreme and high categories.
- **Strengths**:
    - Excels in handling varied data types and distributions.
    - Often provides superior performance due to sequential corrections of errors.
- **Weaknesses**:
    - Can be prone to overfitting if not carefully tuned.
    - More parameters to tune compared to RandomForest, which can increase complexity.

4.3 MLPClassifier

- **Performance Metrics**: Accuracy of 32.16%
- **Confusion Matrix Analysis**:
    - Poor performance with no predictions for the extreme category, indicating issues with model capacity or training.
    - High recall but low precision for the high category, suggesting a bias towards predicting this class.
- **Strengths**:
    - Potential to capture complex, nonlinear relationships in data.
    - Flexible in terms of architecture and can be customized extensively.
- **Weaknesses**:
    - Requires careful tuning of layers, neuron counts, and epochs.
    - Sensitive to feature scaling and the initial random weights.
    - Typically requires more data to perform well compared to tree-based models.

Best Model for the Dataset

- **XGBClassifier** emerges as the best model for this dataset with the highest accuracy and balanced performance across all classes. It effectively handles the categorization task and provides the most reliable predictions among the three tested models.

Conclusion:

- XGBClassifier is recommended due to its robust performance and ability to manage both bias and variance effectively. However, it requires careful hyperparameter tuning and monitoring to prevent overfitting.
- RandomForestClassifier could be considered a close second, especially in scenarios where interpretability and model simplicity are prioritized.
- MLPClassifier would require significantly more tuning and potentially more data or features to be competitive with tree-based models. It shows underperformance in this scenario, likely due to inadequate training or suboptimal architecture.

Recommendaions:

- For deployment, **XGB Classifier** should be fine-tuned further using techniques like grid search or randomized search to optimize its hyperparameters.
- Continuous monitoring and model updates are advised as new data becomes available or as the patterns in bike demand evolve.
- Consider revisiting **MLP Classifier** with adjustments to its architecture or more extensive training, and ensure all input features are scaled appropriately to improve its performance.

5 Conclusion

In conclusion, this study highlights the efficacy of machine learning models in predicting bike rental demand, with XGBoost emerging as the superior classifier due to its highest accuracy and balanced precision-recall scores across all demand categories. The analysis underscores the critical role of feature selection and data preprocessing in enhancing model performance. While Random Forest showed robust performance, particularly in handling feature interactions, it fell short of XGBoost's overall accuracy. MLP, despite its potential for capturing complex nonlinear relationships, underperformed, likely due to its sensitivity to feature scaling and data volume requirements. This case study demonstrates the potential of advanced classification techniques in supporting urban planning and optimizing bike-sharing services, suggesting that future work could focus on further refining models and exploring additional features to enhance predictive accuracy.