IE 7275 Data Mining in Engineering

# Comparative Analysis of Supervised Machine Learning Algorithms

## WITH BIKE SHARING DEMAND DATASET

By - Tejesvani Muppara Vijayaram

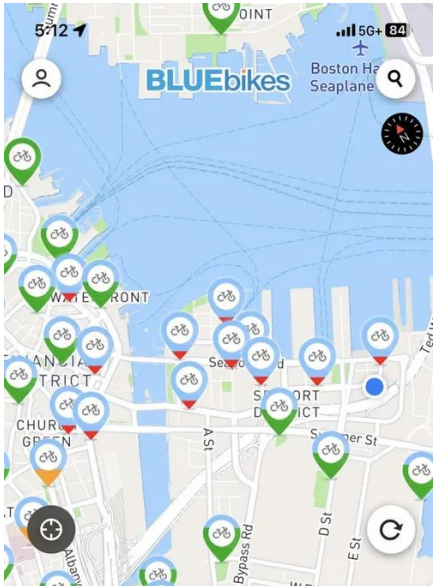
# Table of Contents

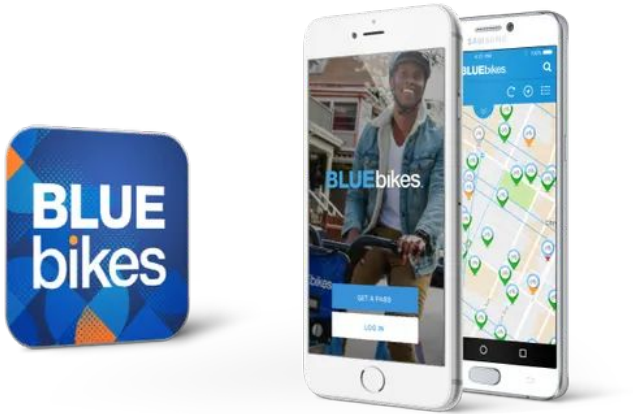
# Problem Statement

The project focuses on leveraging machine learning to analyze and predict bike rental demand.

This will help in gaining the following insights in the Bike Sharing System -

1. Optimize resource allocation
2. Guide system expansion, and
3. Tailor marketing strategies



# Dataset

**Source:** UCI Machine Learning Repository - Dataset

Dataset comprises of 12 features that play a role in determining the demand for the bikes at any given hour of the day.

- **Datetime:** Date and timestamp of the rental
- **Season:** Categorical variable indicates the season
- **Workingday**: Indicating a weekday or a weekend/holiday
- **Holiday**: Whether the day is a holiday or not
- **Weather**: Weather conditions during the rental hour (e.g., clear, cloudy, mist).
- **Temp**: Temperature in Celsius at the time of rental
- **Atemp**: Numerical feature represents the "feels like" temperature in Celsius at the time of rental
- **Humidity**: Gives the relative humidity at the time of rental
- **Windspeed**: Wind speed at the time of rental.
- **Casual**: Number of bike rentals by non-registered users during the hour.
- **Registered**: Number of bike rentals by registered users during the hour.
- **Count**: Total number of bike rentals (casual + registered) during the hour. This is the target variable.


# Preprocessing and understanding the Summary Statistics of the Dataset

1.Check for NULL values

```
# Checking for null values
df.isnull().sum()
```

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

2. Understanding the features in the dataset

```
# Overview of the datatypes of different columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  object
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
 11  count       10886 non-null  int64
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

3. Summary Statistics of Dataset

```
df.describe()
```

| | season | holiday | workingday | weather | temp | atemp | |
|---|---|---|---|---|---|---|---|
| count | 10886.000000 | 10886.000000 | 10886.000000 | 10886.000000 | 10886.00000 | 10886.000000 | 1 |
| mean | 2.506614 | 0.028569 | 0.680875 | 1.418427 | 20.23086 | 23.655084 | |
| std | 1.116174 | 0.166599 | 0.466159 | 0.633839 | 7.79159 | 8.474601 | |
| min | 1.000000 | 0.000000 | 0.000000 | 1.000000 | 0.82000 | 0.760000 | |
| 25% | 2.000000 | 0.000000 | 0.000000 | 1.000000 | 13.94000 | 16.665000 | |
| 50% | 3.000000 | 0.000000 | 1.000000 | 1.000000 | 20.50000 | 24.240000 | |
| 75% | 4.000000 | 0.000000 | 1.000000 | 2.000000 | 26.24000 | 31.060000 | |
| max | 4.000000 | 1.000000 | 1.000000 | 4.000000 | 41.00000 | 45.455000 | |

# Feature Engineering

4. Convert datetime feature to datetime format

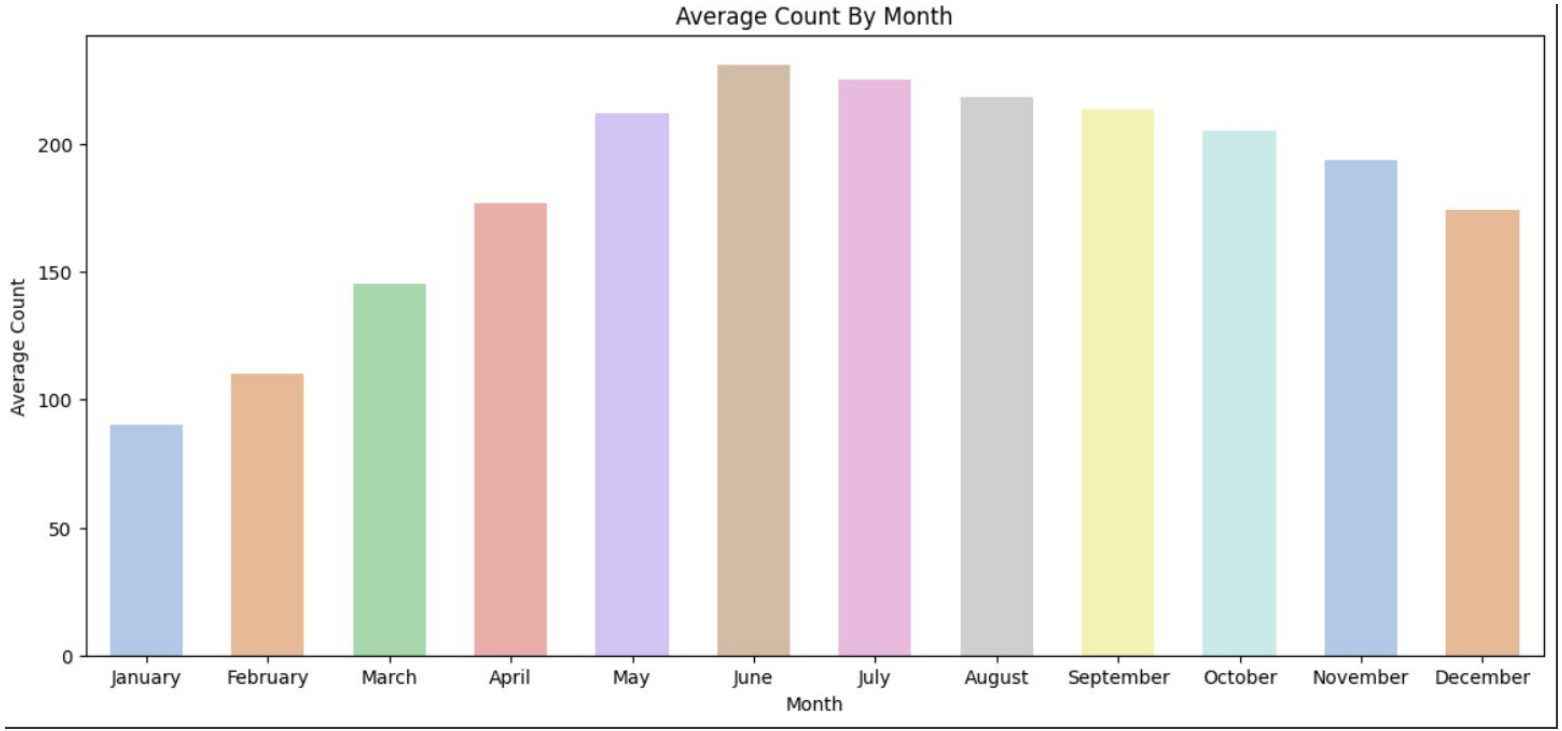
```
#Converting the datetime feature to datetime format
df_updated['datetime'] = pd.to_datetime(df_updated['datetime'])
```

5. Extracting temporal values from datetime column

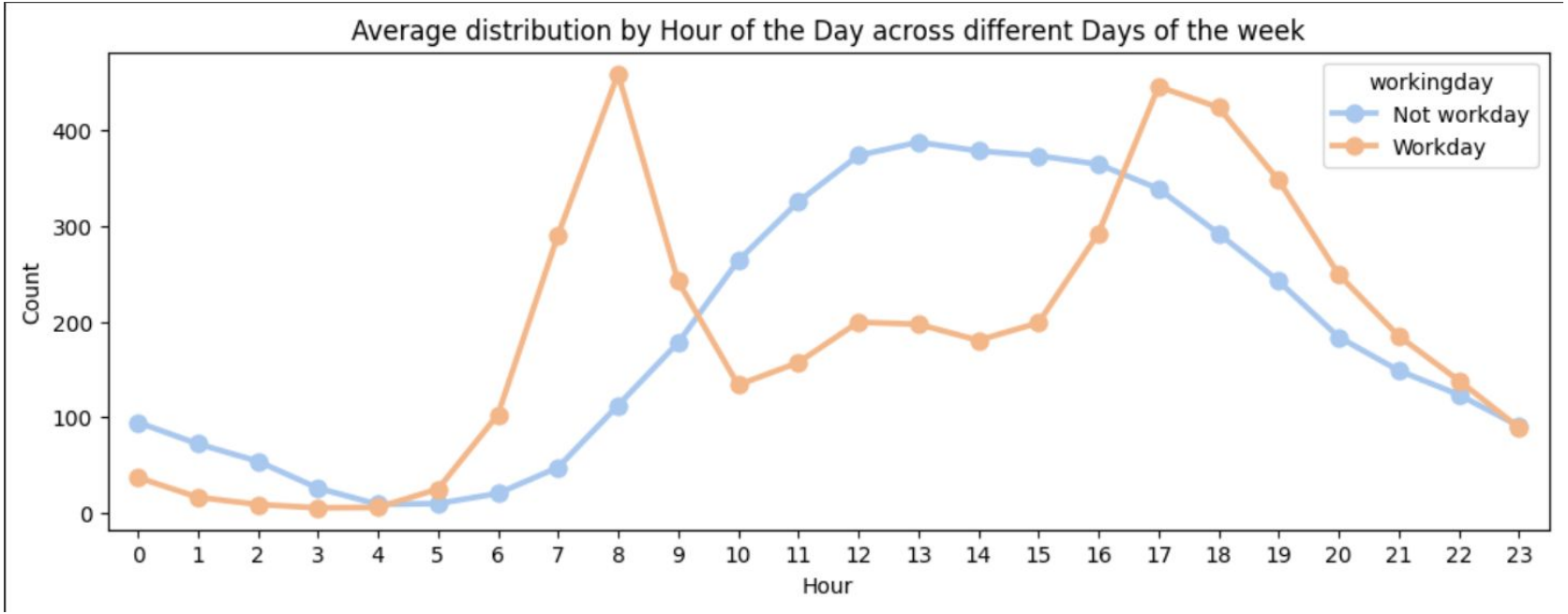
```
# Extracting Date, Year, Month, Day of the week, Time, Hour
df_updated['date'] = df_updated['datetime'].dt.date
df_updated['year'] = df_updated['datetime'].dt.year
df_updated['month'] = df_updated['datetime'].dt.month

# Monday=0 to Sunday=6
df_updated['weekday'] = df_updated['datetime'].dt.weekday

df_updated['time'] = df_updated['datetime'].dt.time
df_updated['hour'] = df_updated['datetime'].dt.hour
```

# Exploratory Data Analysis

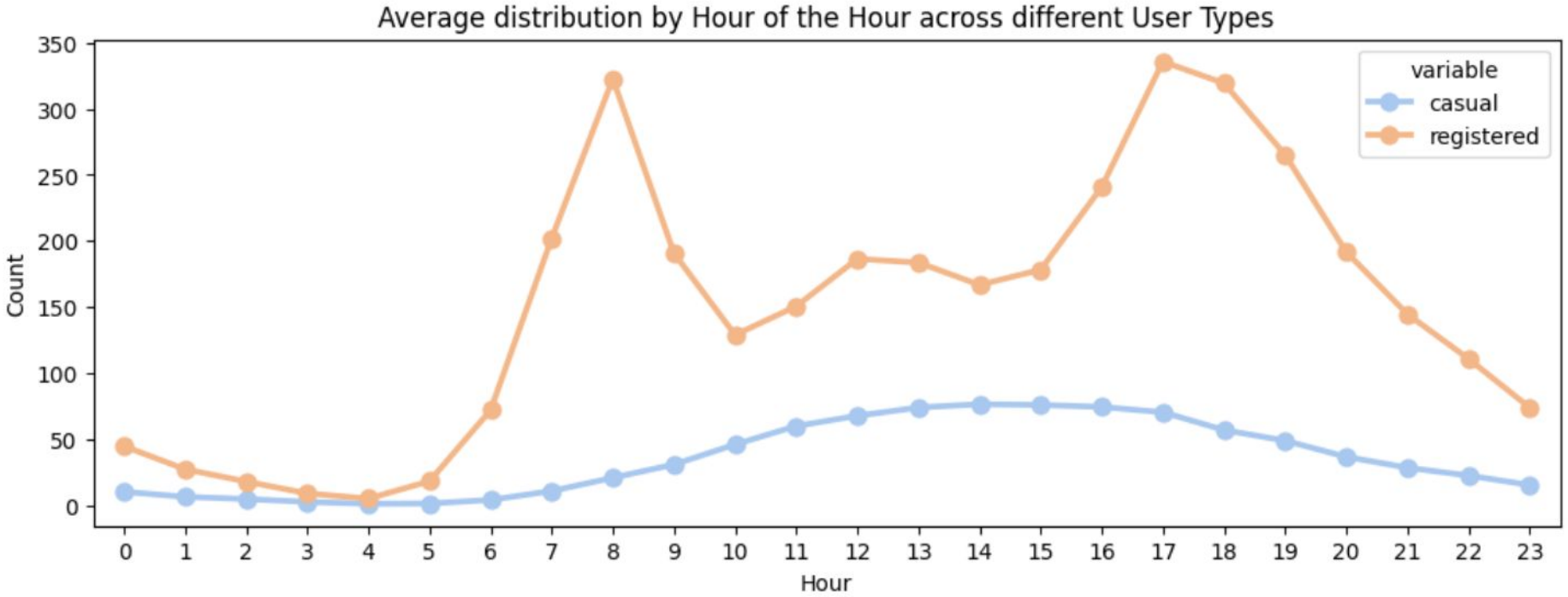
1. **Seasonal Analysis:** Fluctuations in Bike Demand Across the Months of the Year



**2. Hourly Distribution of Bike Demand Across Days of the Week:** Exploring Weekly Patterns in Bike Usage Throughout the Day



**3. Hourly Distribution of Bike Demand Across User Types:** Analyzing Usage Patterns Throughout the Day for Different User Categories

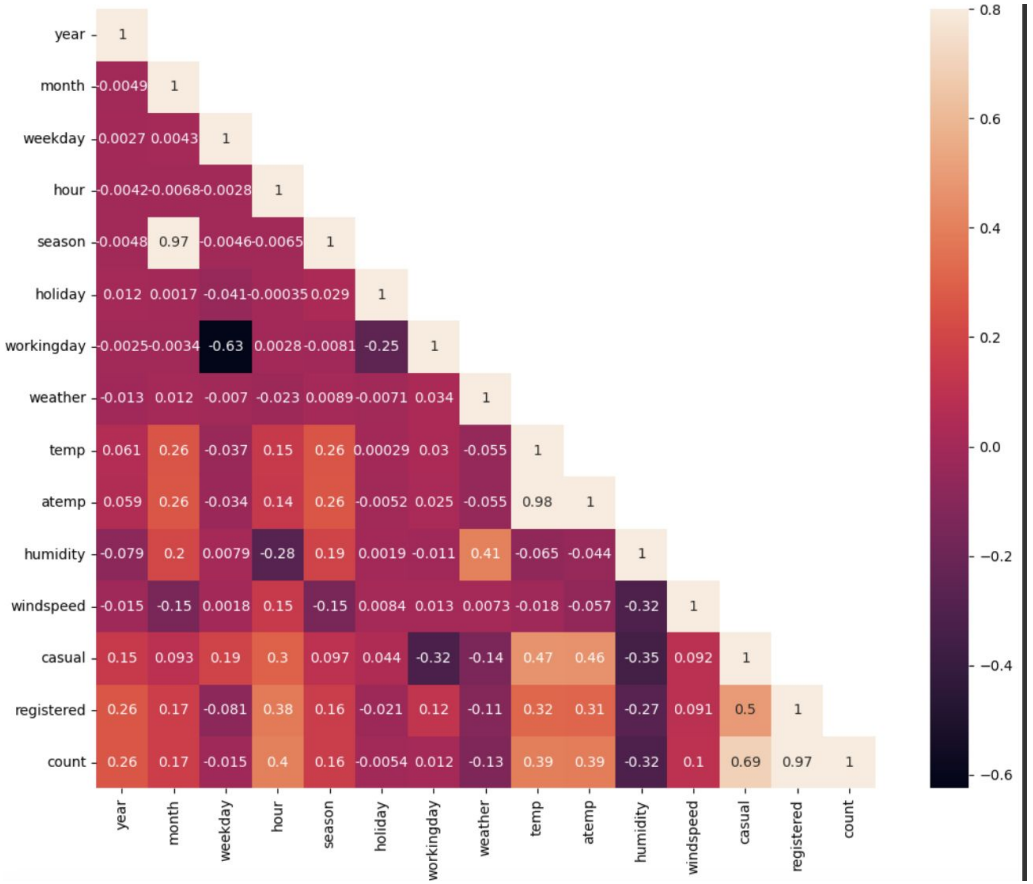


# Correlation Analysis

**Positive Correlations:** The count of bike rentals shows positive correlations with variables such as **year, month, hour, season, temperature** (temp and atemp), and **windspeed**. This suggests that these factors tend to increase along with the count of bike rentals.

**Negative Correlations:** On the other hand, the count of bike rentals demonstrates negative correlations with variables such as **weather** and **humidity**. This indicates that as weather conditions worsen or humidity increases, the count of bike rentals tends to decrease.

**High Correlations:** Variables such as **temp and atemp** exhibit a very high positive correlation, which is expected since they represent temperature-related measurements. Similarly, **seasons and months** show a high positive correlation, reflecting the seasonal variations. As a result the following features show multicollinearity.



# Feature Selection and Encoding for Model Training

Implementing one-hot encoding of weather to enhance model interpretability

```
train_df = train_df[train_df['weather'] != 4]
test_df = test_df[test_df['weather'] != 4]

weather=pd.get_dummies(train_df['weather'],prefix='weather')
weather = weather.apply(lambda x: x.astype(int))
train_df=pd.concat([train_df,weather],axis=1)
train_df.head()
```

Dropping of features that exhibited strong multicollinearity

```
train_df.drop(['datetime','weather','casual','registered'], axis=1, inplace=True)
```

```
train_df = train_df[['weather_1','weather_2','weather_3', 'atemp', 'humidity', 'windspeed', 'hour', 'month', 'year', 'count']]
```

# Hold-out Validation

Hold-out validation using test-train split involves dividing the dataset into two subsets: one for training the model (training set) and the other for evaluating its performance (test set). This approach allows the model to learn patterns from the training data while providing an independent dataset for assessing its generalization ability and performance on unseen data.

```
features_list = train_df.columns.to_list()[:-1]
target = 'count'

X = train_df[features_list]
y = train_df[target]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=40)
```

# Model Selection and Hyperparameter Tuning

Model selection involves choosing the best algorithm or combination of algorithms for a specific predictive modeling problem.

Hyperparameter tuning is the process of optimizing the hyperparameters of a machine learning algorithm to improve its performance.

Grid search is a hyperparameter tuning technique used for systematically searching for the optimal combination of hyperparameters by evaluating the model's performance across a predefined grid of hyperparameter values

The following models were used for this project

- k- Nearest Neighbours
- Decision Tree
- Random Forest
- Gradient Boosting

# K - Nearest Neighbors

KNN (K-Nearest Neighbors) for regression is a simple yet powerful non-parametric algorithm used for making predictions based on the similarity of input features. In regression tasks, KNN calculates the average (or weighted average) of the target values of the k-nearest neighbors to predict the target value for a new data point.

## Base Model Performance

```
Parameters for KNN Regression:
Number of Neighbors (k): 5
Mean Absolute Error (MAE): 82.81010564997702
Mean Squared Error (MSE): 15111.372090032153
Root Mean Squared Error (RMSE): 122.92832094367901
R-squared (R2): 0.5513506417069289
```

# Standardization

Standardization is significant for KNN because the algorithm computes distances between data points to identify nearest neighbors. Features need to be on the same scale to equalizes the impact of all features on the distance calculation.

```
scaler = StandardScaler()

# Define the columns to be standardized
columns_to_standardize = ['atemp', 'humidity', 'windspeed', 'hour', 'month', 'year']

# Fit the scaler on the training data and transform it
train_df_scaled = train_df.copy()
train_df_scaled[columns_to_standardize] = scaler.fit_transform(train_df_scaled[columns_to_standardize])
```

## Performance after Standardization

```
Mean Squared Error (MSE) for scaled data: 12511.33003215434
R-squared (R2) for scaled data: 0.6332162574445132
```

# Hyperparameter Tuning for KNN

**n_neighbors**: Specifies the number of neighbors to consider for prediction

**p**: power parameter for the Minkowski distance metric.

p=1 refers Manhattan distance (L1 norm), and p=2, is Euclidean distance (L2 norm).

**weights**: weight assigned to each neighbor during prediction.

Uniform: equal weight to all neighbors;  Distance: giving closer neighbors more influence

## Performance after Hyperparameter Tuning

```
Best hyperparameters with scaled data: {'n_neighbors': 9, 'p': 1, 'weights': 'distance'}
Mean Absolute Error (MAE) with scaled data: 72.09127649150015
Mean Squared Error (MSE) with scaled data: 11598.103610131744
Root Mean Squared Error (RMSE) with scaled data: 107.69449201389895
R-squared with scaled data: 0.6599885193870212
```

# Decision Tree

Decision Tree for regression is a versatile machine learning algorithm used for predicting continuous outcomes. It divides the input space into regions based on feature values, fitting simple models to each region. This method is intuitive and interpretable, capturing complex relationships between features and the target variable.

## Base Model Performance

```
Decision Tree Regression
Mean Absolute Error (MAE): 68.15112540192926
Mean Squared Error (MSE): 13294.940514469454
Root Mean Squared Error (RMSE): 115.30368820844134
R-squared (R2): 0.6052796202208677
```

# Hyperparameter Tuning for Decision Tree

**max_depth**: Controls the maximum depth of the decision tree, limiting its complexity.

**min_samples_leaf**: Defines the minimum number of samples required to be at a leaf node

**min_samples_split**: Sets the minimum number of samples required to split an internal node

## Performance after Hyperparameter Tuning

```
Best hyperparameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 10}
MAE (Tuned): 61.864673350529465
MSE (Tuned): 8688.288664788439
RMSE (Tuned): 93.21099004295813
R-squared (Tuned): 0.7420488946405129
```

# Random Forest

Random Forest for regression is an ensemble learning method that constructs multiple decision trees during training and outputs the average prediction of the individual trees for regression tasks. It mitigates overfitting and increases accuracy by aggregating the predictions of multiple decision trees.

## Base Model Performance

```
Mean Squared Error (MSE): 7091.77572598722
Root Mean Squared Error (RMSE): 84.21268150336516
Mean Absolute Error (MAE): 54.35899529715423
R-squared (R2): 0.7894485947625307
```

# Hyperparameter Tuning for Random Forest

**n_estimators**: Number of trees in the forest

**max_depth**: Maximum depth of the trees

**min_samples_split**: Minimum number of samples required to split an internal node

**min_samples_leaf**: Minimum number of samples required to be at a leaf node

## Performance after Hyperparameter Tuning

```
Best hyperparameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 150}
Mean Absolute Error (MAE): 56.182771982007154
Mean Squared Error (MSE): 7286.953520589
Root Mean Squared Error (RMSE): 85.36365456439292
R-squared (R2): 0.7836538600568115
```

# Gradient Boosting

Gradient Boosting Regression is an ensemble learning technique that sequentially builds a series of decision trees, each one correcting the errors of its predecessor. It optimizes a loss function by fitting each new tree to the residuals of the previous one, gradually improving predictive accuracy. This iterative process results in a strong predictive model capable of capturing complex relationships in the data.

## Base Model Performance

```
Gradient Boosting Regressor
Mean Absolute Error (MAE): 60.69129377378117
Mean Squared Error (MSE): 7884.409964182894
Root Mean Squared Error (RMSE): 88.79420005936701
R-squared (R2): 0.7659156660378003
```

# Hyperparameter for Gradient Boosting

**n_estimators:** Number of boosting stages to be run

**learning_rate:** Step size shrinkage used in update to prevent overfitting

**max_depth:** Maximum depth of the individual estimators

## Performance after Hyperparameter Tuning

```
Gradient Boosting Regressor (Tuned)
Best Hyperparameters: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100}
Mean Absolute Error (MAE): 57.743901447451975
Mean Squared Error (MSE): 7300.467218310078
Root Mean Squared Error (RMSE): 85.44277159777812
R-squared (R2): 0.7832526448809422
```

# Comparative Analysis

Before Parameter Tuning

| model | mae | mse | rmse | r2 |
|---|---|---|---|---|
| KNN Regression | 82.810106 | 15111.372090 | 122.928321 | 0.551351 |
| KNN Regression with Scaled Data | 75.722948 | 12789.918161 | 113.092520 | 0.625049 |
| Decision Tree Regression | 68.151125 | 13294.940514 | 115.303688 | 0.605280 |
| RandomForestRegressor | 54.358995 | 7091.775726 | 84.212682 | 0.789449 |
| Gradient Boosting Regressor | 60.691294 | 7884.409964 | 88.794200 | 0.765916 |

After Parameter Tuning

| model | mae | mse | rmse | r2 |
|---|---|---|---|---|
| KNN Regression (Tuned) with Scaled Data | 72.091276 | 11598.103610 | 107.694492 | 0.659989 |
| Decision Tree Regression (Tuned) | 61.864673 | 8688.288665 | 93.210990 | 0.742049 |
| RandomForestRegressor (Tuned) | 56.182772 | 7286.953521 | 85.363655 | 0.783654 |
| Best Gradient Boosting Regressor(Tuned) | 57.743901 | 7300.467218 | 85.442772 | 0.783253 |

# Conclusion

The Random Forest Regressor (untuned) achieved the lowest MAE of 54.359, indicating the smallest average error in predictions. The Random Forest Regressor (tuned) achieved a similar performance with an MAE of 56.183, suggesting that hyperparameter tuning did not lead to a significant improvement in this case. Although the improvement may not be significant compared to the base model, it still demonstrates the effectiveness of fine-tuning.

Overall, considering its strong performance metrics, interpretability, and ease of implementation, the **Random Forest Regressor (Tuned) emerges as a reliable choice** for predicting bike sharing demand in Boston.

# Thank You