National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics (MSCDAD_JAN24A_O)

Muhammad Abdur Rabb
Student ID: x23237511

School of Computing
National College of Ireland

Supervisor: Dr. David Hamill

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……. Muhammad Abdur Rabb ……

Student ID: ……x23237511…….……

Programme:…… M.Sc. Data Analytics …… **Year:** ……2024…..

Module: …… Research Project ……..……

Lecturer: …… Dr. David Hamill……..……

Submission Due Date: ……12/12/2024……..……

Project Title: …… Dynamic Pricing using Machine Learning for Emerging Ride-on-demand Service ……..……

Word Count: ……1533…… **Page Count:** ……21……..……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Muhammad Abdur Rabb……

Date: ……11/12/2024……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Muhammad Abdur Rabb
x23237511

1 Introduction

This document provides comprehensive instructions about both hardware and software settings and will explain the practical implementation of the research by describing dataset preparation, pre-processing, model building, and evaluation.

2 Hardware and Software Requirements

2.1 Hardware Configuration

This research work was done on a personal laptop and system configuration settings are shown in Figure 1. The hardware configuration is as follows:

- Processor: 13th Gen Intel(R) Core(TM) i5-13500H CPU @ 2.60 GHz
- Installed RAM: 16 GB (15.6 GB usable)
- System Type: 64-bit operating system, x64-based processor

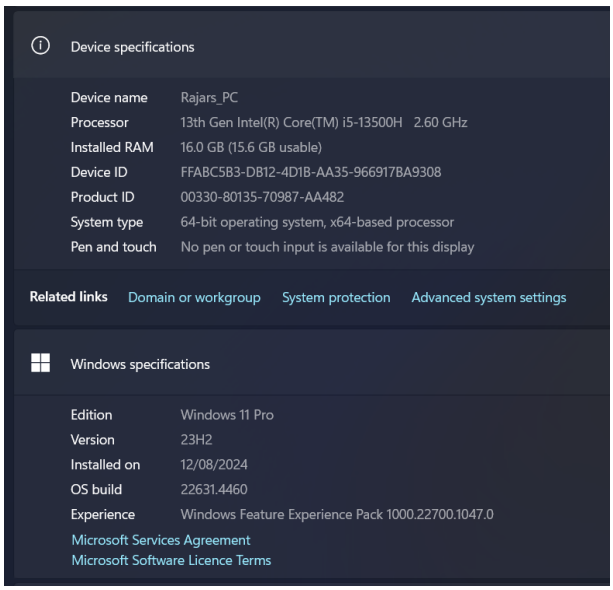


Figure 1: System Configuration

2.2 Software Configuration

This section describes all the environments that were configured and used for the implementation, which should have been ready in advance. Following software or applications have been setup and need to be installed onto the system before going ahead with the process:

- Operating System: Windows 11 Pro (64-bit)
- Development Environment: Visual Studio Code (VSCode) configured with the Jupyter Notebook extension for interactive coding and analysis.
- Programming Language: Python (latest version at the time of development, Python 3.12.5).

It should be underlined at this point that this setting is not mandatory in terms of software. The project may also be executed for other environments, like Google Colab, Anaconda, or similar, including macOS-based ones. In such a case, however, users would be expected to have the environment properly prepared to work with the toolset and workflows concerned.

3 Packages & Libraries

Data analysis and any kind of machine learning require the importing of certain packages and libraries. Figure 2 shows the compilation of libraries taken in use for this project. These should be installed before the actual running of code so that the availability of functions and features can be guaranteed.

The following libraries can be installed by running the following in your terminal or command prompt:

pip install pandas numpy matplotlib seaborn scikit-learn

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
```

Figure 2: Imported Libraries and Modules for the Project

4 Dataset

The dataset used in this analysis can be downloaded from the following Kaggle: https://www.kaggle.com/datasets/yasserh/uber-fares-dataset. After downloading the dataset, it has to be extracted onto the preferred folder in which you plan to code. The extraction must be in such a way that the extracted file is readily accessible from the environment where the coding is being performed because it will be the main dataset under analysis and model training.

The code in Figure 3 shows how the dataset can be loaded into a Pandas DataFrame. It also gives an overview of the dataset after loading, showing the important features of fare_amount, pickup_datetime, pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude, and passenger_count. These are the major features which, in this project, data preprocessing and machine learning modelling will be based.

```
df = pd.read_csv('uber.csv')
df.head()
```

	Unnamed: 0	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	24238194	2015-05-07 19:52:06.0000003	7.5	2015-05-07 19:52:06 UTC	-73.999817	40.738354	-73.999512	40.723217	1
1	27835199	2009-07-17 20:04:56.0000002	7.7	2009-07-17 20:04:56 UTC	-73.994355	40.728225	-73.994710	40.750325	1
2	44984355	2009-08-24 21:45:00.00000061	12.9	2009-08-24 21:45:00 UTC	-74.005043	40.740770	-73.962565	40.772647	1
3	25894730	2009-06-26 08:22:21.0000001	5.3	2009-06-26 08:22:21 UTC	-73.976124	40.790844	-73.965316	40.803349	3
4	17610152	2014-08-28 17:47:00.000000188	16.0	2014-08-28 17:47:00 UTC	-73.925023	40.744085	-73.973082	40.761247	5

Figure 3: Dataset Preview After Loading

This dataset comprises 200,000 rows and 9 columns, as shown in Figure 4 below, which summarizes the data types for each column. Also, checks for missing values, duplicates, and null entries present this as a clean dataset with very little preprocessing required.

```
print(df.shape)
print(df.dtypes)
```

```
(200000, 9)
Unnamed: 0             int64
key                   object
fare_amount          float64
pickup_datetime       object
pickup_longitude     float64
pickup_latitude      float64
dropoff_longitude    float64
dropoff_latitude     float64
passenger_count        int64
dtype: object
```

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

```
Unnamed: 0           0
key                  0
fare_amount          0
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
dropoff_longitude    1
dropoff_latitude     1
passenger_count      0
dtype: int64
```

```
df.duplicated().sum()
```

```
0
```

Figure 4: Dataset Shape, Data Types, Missing Values, and Duplicate Check

5 Exploratory Data Analysis

5.1 Basic EDA

EDA has been carried out to understand the structure and distribution of the data, locate some outlier cases, and extract insights over some important features. The following figures are the code with their respective outputs that were developed in the EDA conducted on the dataset.

```
plt.figure(figsize=(7, 2))
plt.title('Distance in KM', fontsize=14, fontweight='bold')
sns.boxplot(data=df1, x='ride_distance', fliersize=1, color='skyblue')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

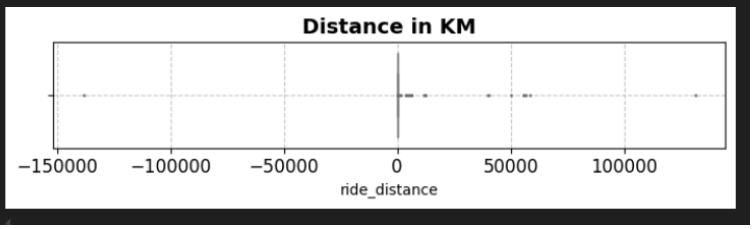


Figure 5: Boxplot of Ride Distance

```
plt.figure(figsize=(7, 3))
sns.histplot(df1['ride_distance'], bins=range(0, 30, 1), color='skyblue')
plt.title('Distance Histogram', fontsize=14, fontweight='bold')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

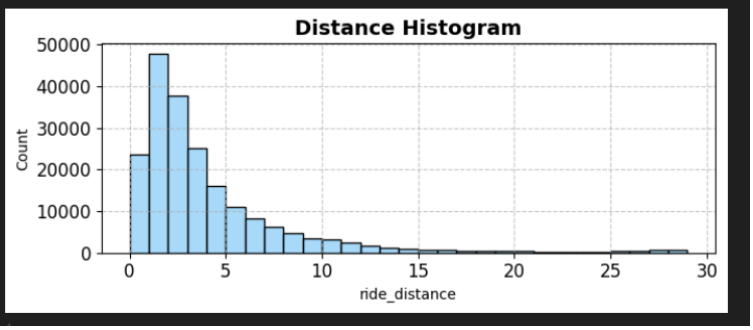


Figure 6: Histogram of Ride Distance

```
plt.figure(figsize=(7, 2))
plt.title('Fare Amount', fontsize=14, fontweight='bold')
sns.boxplot(data=df1, x='fare_amount', fliersize=1, color='skyblue')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

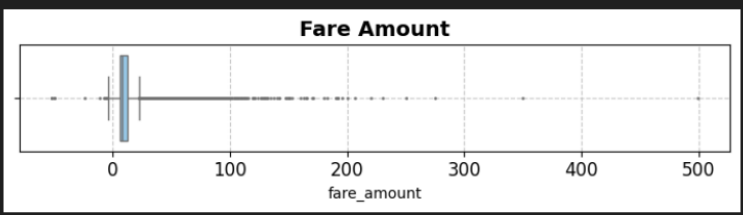


Figure 7: Boxplot of Fare Amount

```
plt.figure(figsize=(7, 3))
sns.histplot(df1['fare_amount'], bins=range(0, 30, 1), color='skyblue')
plt.title('Fare Amount Histogram', fontsize=14, fontweight='bold')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

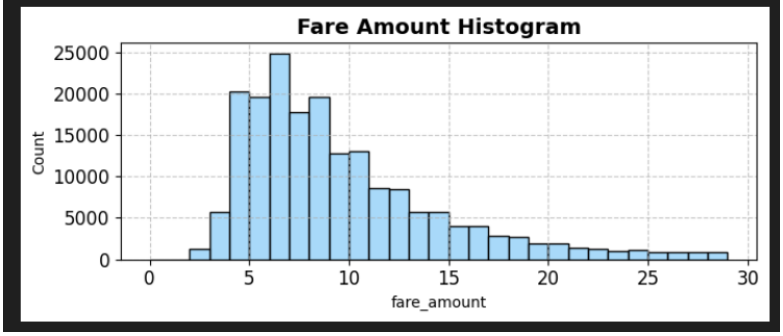


Figure 8: Histogram of Fare Amount

```
low_fare_count = df1[df1['fare_amount'] < 4].shape[0]
high_fare_count = df1[df1['fare_amount'] > 130].shape[0]

plt.figure(figsize=(6, 4))
ax1 = sns.barplot(x=['Incorrect Fares (<$4)', 'Outlier Fares (>$130)'],
                  y=[low_fare_count, high_fare_count], palette='Set2',
                  hue=['Low Fares (<$4)', 'High Fares (>$130)'], legend=False)
for p in ax1.patches:
    ax1.annotate(format(p.get_height(), '.0f'),
                 (p.get_x() + p.get_width() / 2., p.get_height()),
                 ha = 'center', va = 'bottom',
                 fontsize=12, color='black')
plt.title('Count of Low and High Fares')
plt.ylabel('Count')
plt.show()
```

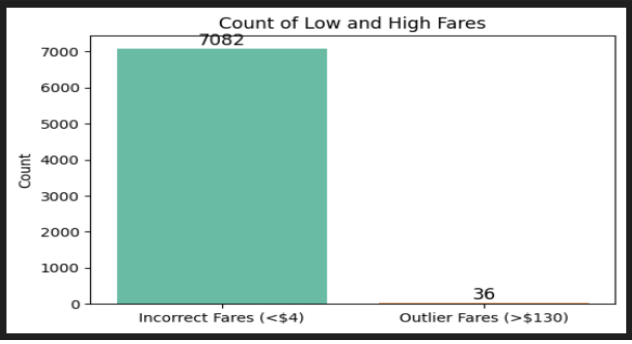


Figure 9: Bar Plot Showing Count of Low and High Fares

```
plt.figure(figsize=(7, 3))
sns.histplot(df1['passenger_count'], bins=range(0, 30, 1), color='skyblue')
plt.title('Passenger Count Histogram', fontsize=14, fontweight='bold')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

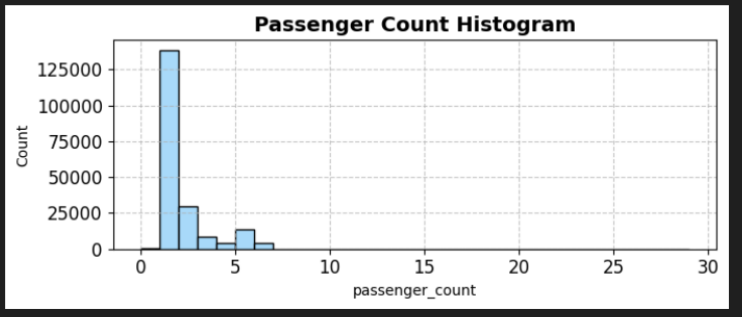


Figure 10: Histogram of Passenger Count

```
plt.figure(figsize=(7, 3))

ax = sns.countplot(data=df1, x='passenger_count', palette='Set2', hue='passenger_count', legend=False)
for p in ax.patches:
    ax.annotate(format(p.get_height(), '.0f'),
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha = 'center', va = 'center',
                xytext = (0, 10),
                textcoords = 'offset points')
plt.show()
```

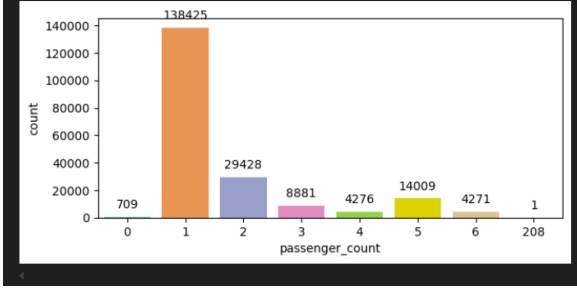


Figure 11: Count Plot of Passenger Count

```
columns = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude']
titles = ['Pickup Longitude', 'Pickup Latitude', 'Dropoff Longitude', 'Dropoff Latitude']
plt.figure(figsize=(12, 6))
for i, (column, title) in enumerate(zip(columns, titles), 1):
    plt.subplot(2, 2, i)
    sns.boxplot(data=df, x=column)
    plt.title(f'Boxplot of {title}')
plt.tight_layout()
plt.show()
```

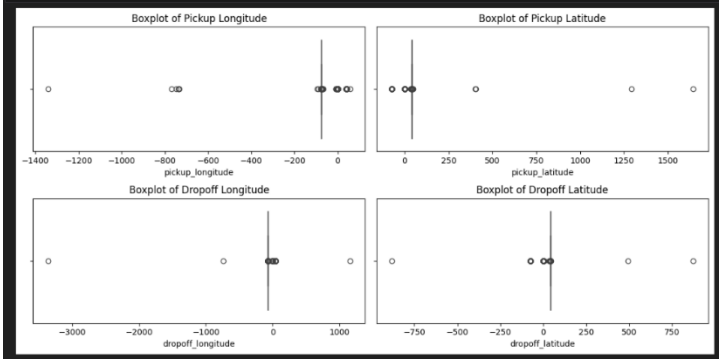


Figure 12: Boxplots of Pickup and Dropoff Coordinates

```
nyc_lat_min, nyc_lat_max = 40.4774, 40.9176
nyc_lon_min, nyc_lon_max = -74.2591, -73.7004
plt.figure(figsize=(10, 6))
plt.scatter(df['pickup_longitude'], df['pickup_latitude'], s=10, c='blue', label='Pickups', alpha=0.5)
plt.scatter(df['dropoff_longitude'], df['dropoff_latitude'], s=10, c='red', label='Dropoffs', alpha=0.5)
plt.title('Scatter Plot of Pickup and Dropoff Coordinates')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()

outside_nyc_pickups = df[
    (df['pickup_latitude'] < nyc_lat_min) | (df['pickup_latitude'] > nyc_lat_max) |
    (df['pickup_longitude'] < nyc_lon_min) | (df['pickup_longitude'] > nyc_lon_max)
]

outside_nyc_dropoffs = df[
    (df['dropoff_latitude'] < nyc_lat_min) | (df['dropoff_latitude'] > nyc_lat_max) |
    (df['dropoff_longitude'] < nyc_lon_min) | (df['dropoff_longitude'] > nyc_lon_max)
]

pickup_outside_count = len(outside_nyc_pickups)
dropoff_outside_count = len(outside_nyc_dropoffs)

pickup_outside_count, dropoff_outside_count
```

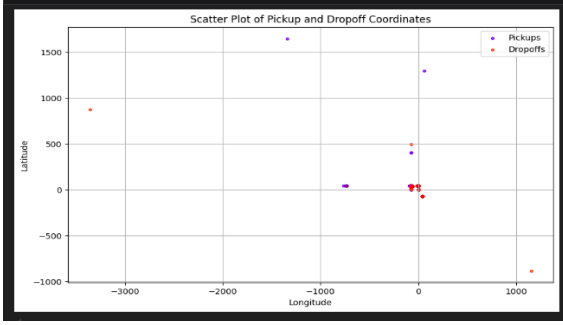


Figure 13: Scatter Plot of Pickup and Dropoff Coordinates

5.2 Advanced EDA

The following sections are a series of visualizations that tried answering the research questions by analysing trends and patterns in this dataset. The following plots examine the relationship among some of the major variables: average fare amount against hour of day, day of week, and passenger count. In addition, scatter plots on such relationships as fare amount against ride distance and passenger count. Further, distance categories are analysed against fare amounts to highlight the differences in pricing patterns.

The following figures show the code and their respective outputs for this analysis.

```
avg_fare_by_hour = df2.groupby('Hour')['fare_amount'].mean()
plt.figure(figsize=(10, 6))
avg_fare_by_hour.plot(kind='bar', color='purple')
plt.title('Average Fare Amount by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Average Fare Amount')
plt.xticks(rotation=0)
plt.show()
```

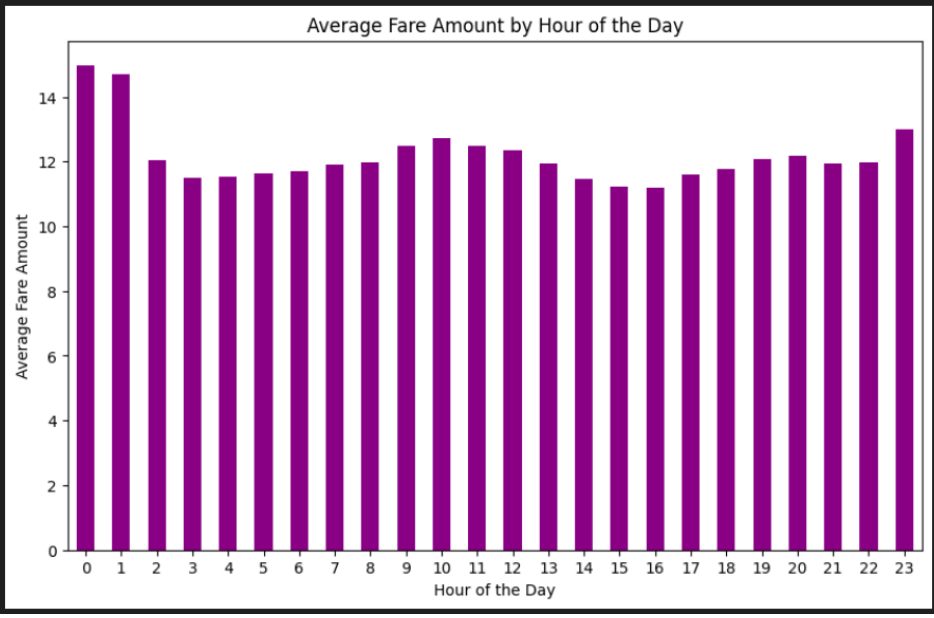


Figure 14: Average Fare Amount by Hour of the Day

```
rides_per_hour = df2.groupby('Hour')['fare_amount'].count()
plt.figure(figsize=(10, 6))
rides_per_hour.plot(kind='bar', color='green')
plt.title('Number of Rides by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Rides')
plt.xticks(rotation=0)
plt.show()
```

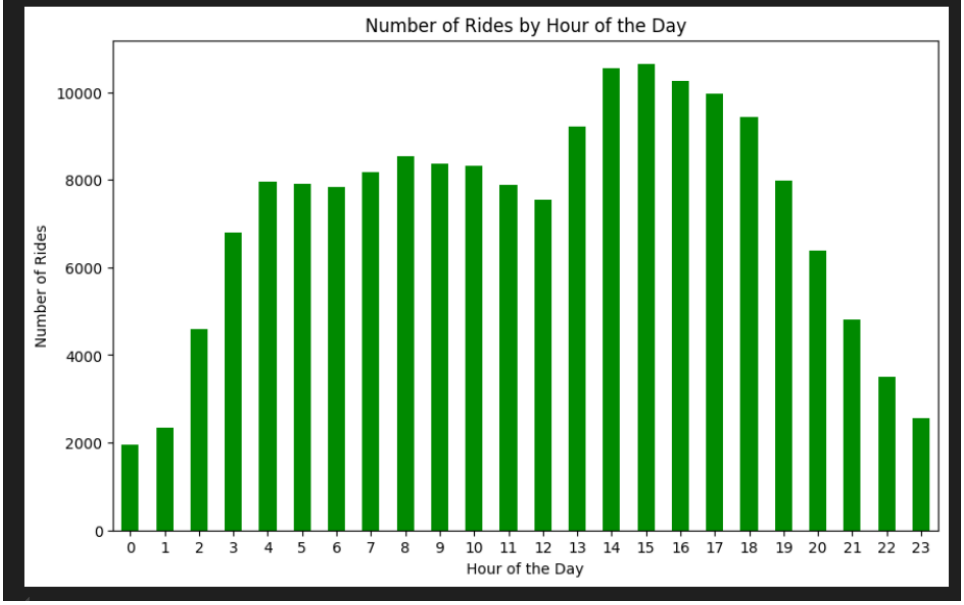


Figure 15: Number of Rides by Hour of the Day

```
avg_fare_by_weekday = df2.groupby('WeekDay')['fare_amount'].mean()
plt.figure(figsize=(10, 6))
avg_fare_by_weekday.plot(kind='bar', color='purple')
plt.title('Average Fare Amount by Weekday')
plt.xlabel('Day of the Week (0=Sunday, 6=Saturday)')
plt.ylabel('Average Fare Amount')
plt.xticks(rotation=0)
plt.show()
```

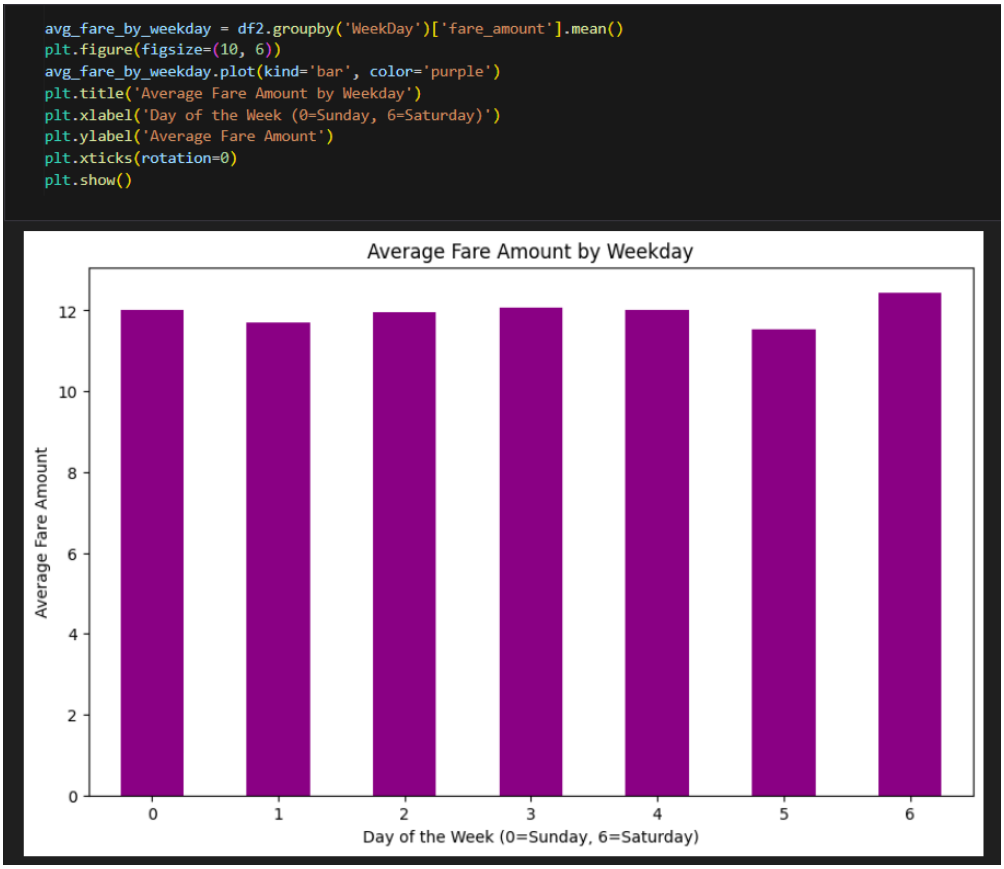


Figure 16: Average Fare Amount by Weekday

```
rides_per_weekday = df2.groupby('WeekDay')['fare_amount'].count()
plt.figure(figsize=(10, 6))
rides_per_weekday.plot(kind='bar', color='green')
plt.title('Number of Rides by Weekday')
plt.xlabel('Day of the Week (0=Sunday, 6=Saturday)')
plt.ylabel('Number of Rides')
plt.xticks(rotation=0)
plt.show()
```

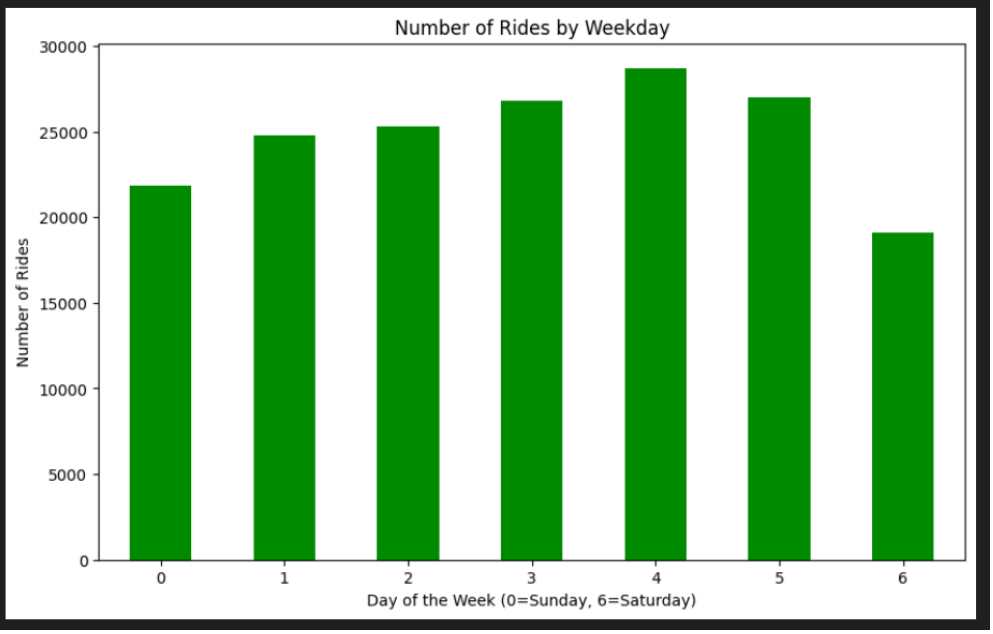


Figure 17: Number of Rides by Weekday

```
plt.figure(figsize=(8, 6))
plt.scatter(df2['ride_distance'], df2['fare_amount'], alpha=0.5, color='blue')
plt.title('Ride Distance vs Fare Amount')
plt.xlabel('Ride Distance (miles)')
plt.ylabel('Fare Amount ($)')
plt.show()

correlation_distance_fare = df2[['ride_distance', 'fare_amount']].corr()

correlation_distance_fare
```

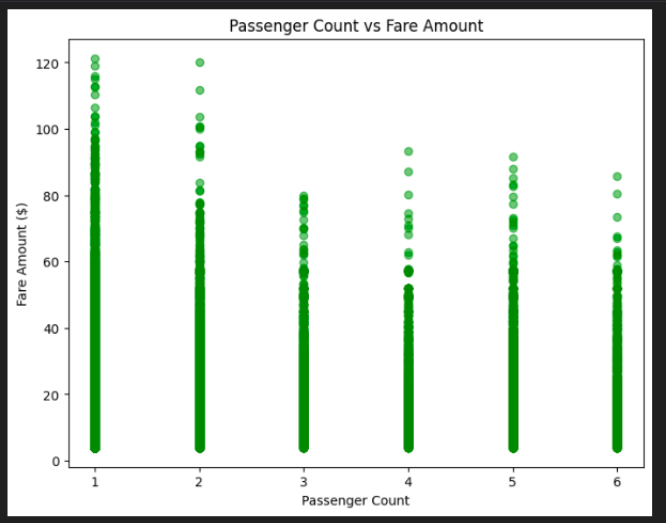
	ride_distance	fare_amount
ride_distance	1.000000	0.890602
fare_amount	0.890602	1.000000

Figure 17: Ride Distance vs Fare Amount

```
plt.figure(figsize=(8, 6))
plt.scatter(df2['passenger_count'], df2['fare_amount'], alpha=0.5, color='green')
plt.title('Passenger Count vs Fare Amount')
plt.xlabel('Passenger Count')
plt.ylabel('Fare Amount ($)')
plt.show()

correlation_passenger_fare = df2[['passenger_count', 'fare_amount']].corr()

correlation_passenger_fare
```



	passenger_count	fare_amount
passenger_count	1.000000	0.013643
fare_amount	0.013643	1.000000

Figure 18: Passenger Count vs Fare Amount

```
bins = [0, 2, 5, 10, 20, df2['ride_distance'].max()]
labels = ['0-2 km', '2-5 km', '5-10 km', '10-20 km', '20+ km']
df2['distance_category'] = pd.cut(df2['ride_distance'], bins=bins, labels=labels)

avg_fare_by_distance_hour = df2.groupby(['Hour', 'distance_category'])['fare_amount'].mean().unstack()

plt.figure(figsize=(10, 6))
avg_fare_by_distance_hour.plot(kind='line', marker='o')
plt.title('Average Fare by Hour for Different Distance Categories')
plt.xlabel('Hour of the Day')
plt.ylabel('Average Fare Amount ($)')
plt.legend(title='Distance Category (km)')
plt.grid(True)
plt.show()

avg_fare_by_distance_hour.head()
```

```
C:\Users\rajar\AppData\Local\Temp\ipykernel_21552\3712950940.py:5: FutureWarning: The default of observed=Fa
  avg_fare_by_distance_hour = df2.groupby(['Hour', 'distance_category'])['fare_amount'].mean().unstack()
<Figure size 1000x600 with 0 Axes>
```

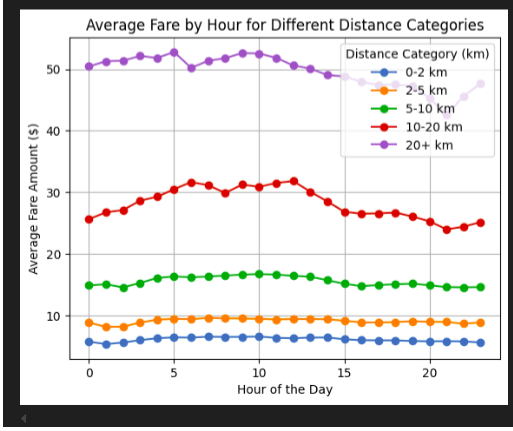


Figure 19: Average Fare by Hour for Distance Categories

```
avg_fare_by_distance_weekday = df2.groupby(['WeekDay', 'distance_category'])['fare_amount'].mean().unstack()

plt.figure(figsize=(10, 6))
avg_fare_by_distance_weekday.plot(kind='line', marker='o')
plt.title('Average Fare by Weekday for Different Distance Categories')
plt.xlabel('Day of the Week (0=Sunday, 6=Saturday)')
plt.ylabel('Average Fare Amount ($)')
plt.legend(title='Distance Category (km)')
plt.grid(True)
plt.show()

avg_fare_by_distance_weekday.head()
```

```
C:\Users\rajar\AppData\Local\Temp\ipykernel_21552\222130875.py:1: FutureWarning: The default of observed=False is
  avg_fare_by_distance_weekday = df2.groupby(['WeekDay', 'distance_category'])['fare_amount'].mean().unstack()
<Figure size 1000x600 with 0 Axes>
```

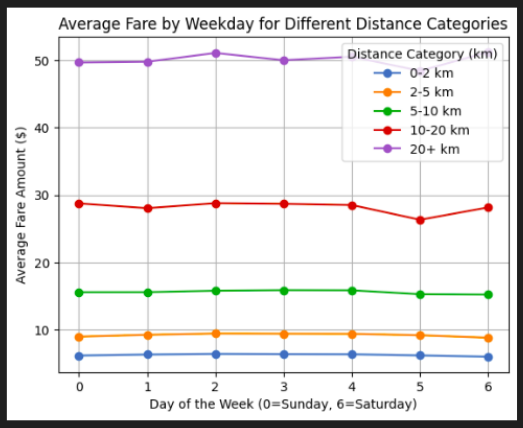


Figure 20: Average Fare by Weekday for Distance Categories

6 Data Preparation

The preprocessing steps include cleaning and feature engineering: cleaning the data by removing unnecessary columns, and feature engineering, creating new features including ride_distance that is computed using a custom Manhattan distance function. Extract temporal features like Hour, WeekDay, and Month from it and clean the invalid or extreme values according to the conditions. Later, this dataset was standardized and cyclic and categorical features encoded to work with the model. The following figures represent the code used for these steps.

```
def manhattan_distance(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = np.radians([lat1, lon1, lat2, lon2])
    km_per_radian = 6371
    d_lat = np.abs(lat2 - lat1)
    d_lon = np.abs(lon2 - lon1)
    lat_distance_km = d_lat * km_per_radian
    lon_distance_km = d_lon * km_per_radian * np.cos((lat1 + lat2) / 2)
    return lat_distance_km + lon_distance_km
df1['ride_distance'] = df1.apply(lambda row: manhattan_distance(
    row['pickup_latitude'], row['pickup_longitude'],
    row['dropoff_latitude'], row['dropoff_longitude']), axis=1)
```

Figure 21: Calculating Ride Distance Using Manhattan Distance Formula

```
df1['pickup_datetime'] = pd.to_datetime(df1['pickup_datetime'])
df1['pickup_datetime'] = df1['pickup_datetime'].dt.tz_convert('America/New_York')

df1['Date'] = df1['pickup_datetime'].dt.date
df1['Time'] = df1['pickup_datetime'].dt.strftime('%H:%M')
df1['Month'] = df1['pickup_datetime'].dt.month
df1['WeekDay'] = df1['pickup_datetime'].dt.weekday
df1['Hour'] = df1['pickup_datetime'].dt.hour
```

Figure 22: Extracting Temporal Features from Pickup Datetime

```
df1.drop(['Unnamed: 0', 'key'], axis=1, inplace=True)
```

Figure 23: Dropping Unnecessary Columns

```
df2 = df1[(df1['fare_amount'] >= 4) & (df1['fare_amount'] <= 130)]


df2 = df2[(df2['ride_distance'] >= 1) & (df2['ride_distance'] <= 80)]


df2 = df2[(df2['passenger_count'] >= 1) & (df2['passenger_count'] <= 6)]
```

Figure 24: Removing Outliers in Fare Amount, Ride Distance, and Passenger Count

```
nyc_longitude_bounds = (-74.25909, -73.700272)
nyc_latitude_bounds = (40.477399, 40.917577)

df2 = df2[
    (df2['pickup_longitude'].between(*nyc_longitude_bounds)) &
    (df2['pickup_latitude'].between(*nyc_latitude_bounds)) &
    (df2['dropoff_longitude'].between(*nyc_longitude_bounds)) &
    (df2['dropoff_latitude'].between(*nyc_latitude_bounds))
]
```

Figure 25: Filtering Entries Within NYC Geographic Boundaries

```
df2.drop(['pickup_datetime', 'pickup_longitude', 'pickup_latitude',
          'dropoff_longitude', 'dropoff_latitude', 'Date', 'Time',
           'distance_category'], axis=1, inplace=True)
df2.head()
```

Figure 26: Dropping Unused Columns After Cleaning

```
scaler = StandardScaler()
df2[['fare_amount', 'ride_distance']] = scaler.fit_transform(df2[['fare_amount', 'ride_distance']])

df2['Hour_sin'] = np.sin(2 * np.pi * df2['Hour'] / 24)
df2['Hour_cos'] = np.cos(2 * np.pi * df2['Hour'] / 24)
df2['WeekDay_sin'] = np.sin(2 * np.pi * df2['WeekDay'] / 7)
df2['WeekDay_cos'] = np.cos(2 * np.pi * df2['WeekDay'] / 7)

df2 = pd.get_dummies(df2, columns=['Month'], prefix='Month')

df2 = df2.drop(columns=['Hour', 'WeekDay'])

df2.head()
```

Figure 27: Standardizing, Encoding Cyclic Features, and One-Hot Encoding

7 Machine Learning Models

Various machine learning models were implemented using features such as ride distance, passenger count, and time-based variables in order to obtain the fare amount in this section. First of all, it splits the data into a training set and a test set. Then, the model training includes Linear Regression, Random Forest, Neural Networks, and Gradient Boosting on this dataset. Each of the models developed was assessed in terms of their MAE, RMSE, and R^2 score that would outline their performance in terms of accuracy and robustness. The best performing, in this respect, is Gradient Boosting, hence quite suitable for this task of prediction.

The following figures show code and result of each model:

```
X = df2.drop(columns=['fare_amount'])
y = df2['fare_amount']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 28: Splitting the Dataset into Training and Testing Sets

Linear Regression (Baseline model)

```
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

y_pred = linear_model.predict(X_test)
```

```
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

mae, rmse, r2
```

```
(0.26268196621934164, 0.4538672063839813, 0.7921077657994645)
```

Figure 29: Linear Regression (Baseline Model) Code and Results

Random Forest

```
random_forest_model = RandomForestRegressor(random_state=42, n_estimators=100)
random_forest_model.fit(X_train, y_train)
y_pred_rf = random_forest_model.predict(X_test)
```

```
mae_rf = mean_absolute_error(y_test, y_pred_rf)
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
r2_rf = r2_score(y_test, y_pred_rf)

mae_rf, rmse_rf, r2_rf
```

```
(0.2730050256525148, 0.46387605262001697, 0.7828376346538732)
```

Figure 30: Random Forest Model Code and Results

Neural Netwroks

```
mlp_model = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=500, random_state=42)
mlp_model.fit(X_train, y_train)

y_pred_mlp = mlp_model.predict(X_test)
```

```
mae_mlp = mean_absolute_error(y_test, y_pred_mlp)
rmse_mlp = np.sqrt(mean_squared_error(y_test, y_pred_mlp))
r2_mlp = r2_score(y_test, y_pred_mlp)

mae_mlp, rmse_mlp, r2_mlp
```

```
(0.2585898013981667, 0.45179801882550563, 0.7939990126098191)
```

Figure 31: Neural Networks (MLP) Model Code and Results

Gradient Boosting

```
gb_model = GradientBoostingRegressor(random_state=42, n_estimators=100)
gb_model.fit(X_train, y_train)

y_pred_gb = gb_model.predict(X_test)
```

```
mae_gb = mean_absolute_error(y_test, y_pred_gb)
rmse_gb = np.sqrt(mean_squared_error(y_test, y_pred_gb))
r2_gb = r2_score(y_test, y_pred_gb)

mae_gb, rmse_gb, r2_gb
```

```
(0.255004868045097, 0.4411343662908811, 0.8036086105533072)
```

Figure 31: Gradient Boosting Model Code and Results

7.1 Model Comparison

After training the machine learning models, a comparison was made between them to evaluate their performance using some key metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score. As evident from the results, Gradient Boosting outperformed all the other models by scoring the lowest in both MAE and RMSE, along with having the highest R^2 score, indicating better predictive power. For each metric, visualizations were also created to give a complete overview of how the models compare.

The next couple of figures show the code and the outputs for model comparison:

Model comparison

```
print("Linear Regression:", mae, rmse, r2)
print("Random Forest:", mae_rf, rmse_rf, r2_rf)
print("MLP:", mae_mlp, rmse_mlp, r2_mlp)
print("Gradient Boosting:", mae_gb, rmse_gb, r2_gb)
```

```
Linear Regression: 0.26268196621934164 0.4538672063839813 0.7921077657994645
Random Forest: 0.2730050256525148 0.46387605262001697 0.7828376346538732
MLP: 0.2585898013981667 0.45179801882550563 0.7939990126098191
Gradient Boosting: 0.255004868045097 0.4411343662908811 0.8036086105533072
```

```
models = ['Linear Regression', 'Random Forest', 'MLP', 'Gradient Boost']
mae_models = [mae, mae_rf, mae_mlp, mae_gb]
rmse_models = [rmse, rmse_rf, rmse_mlp, rmse_gb]
r2_models = [r2, r2_rf, r2_mlp, r2_gb]
```

Figure 32: Code for Model Performance Comparison

```
colors = ['skyblue', 'orange', 'green', 'red']
plt.figure(figsize=(10, 6))
bars = plt.barh(models, mae_models, color=colors, alpha=0.8)

for i, bar in enumerate(bars):
    plt.text(bar.get_width() + 0.01,
             bar.get_y() + bar.get_height() / 2,
             f'{mae_models[i]:.5f}',
             fontsize=10, fontweight='bold', va='center', ha='left')

plt.title('Mean Absolute Error (MAE) of Models', fontsize=14, fontweight='bold')
plt.xlabel('MAE', fontsize=12, fontweight='bold')
plt.ylabel('Model', fontsize=12, fontweight='bold')
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.xticks(fontsize=10, fontweight='bold')
plt.yticks(fontsize=10,fontweight='bold')
plt.tight_layout()
plt.show()
```

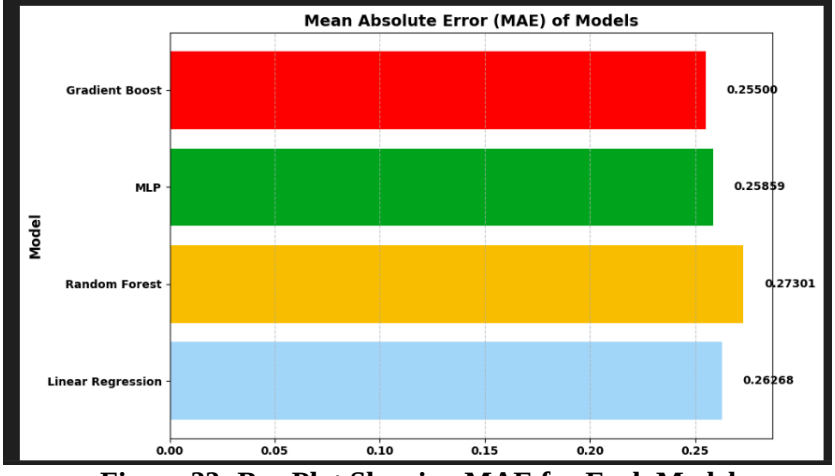


Figure 33: Bar Plot Showing MAE for Each Model

```python
plt.figure(figsize=(10, 6))
bars = plt.bar(models, rmse_models, color=colors, alpha=0.8)

for i, bar in enumerate(bars):
    plt.text(bar.get_x() + bar.get_width() / 2,
             bar.get_height() + 0.01,
             f'{rmse_models[i]:.5f}',
             fontsize=10, fontweight='bold', ha='center', va='bottom')

plt.title('Root Mean Squared Error (RMSE) of Models', fontsize=14, fontweight='bold')
plt.xlabel('Model', fontsize=12, fontweight='bold')
plt.ylabel('RMSE', fontsize=12, fontweight='bold')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.xticks(fontsize=10, fontweight='bold')
plt.yticks(fontsize=10, fontweight='bold')
plt.tight_layout()
plt.show()
```

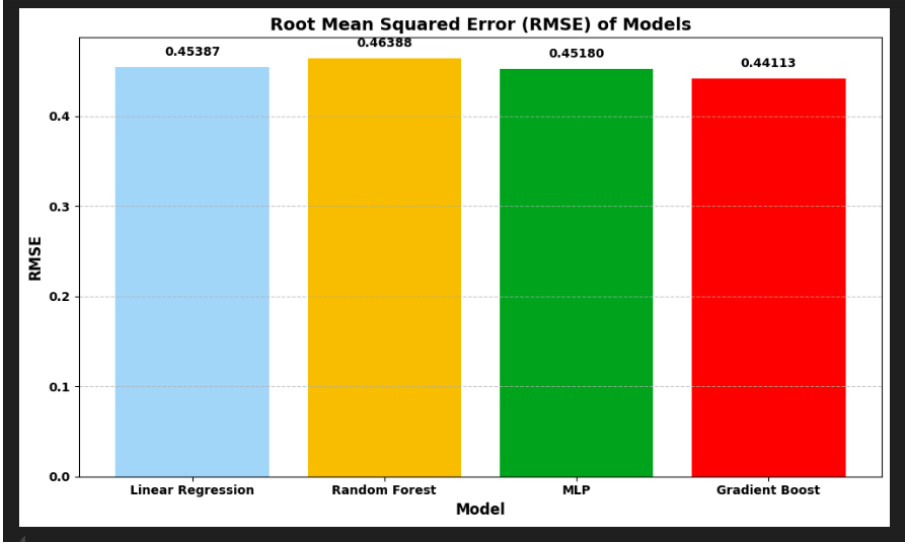


Figure 34: Bar Plot Showing RMSE for Each Model

```python
plt.figure(figsize=(10, 6))
bars = plt.barh(models, r2_models, color=colors, alpha=0.8)

for i, bar in enumerate(bars):
    plt.text(bar.get_width() + 0.01,
             bar.get_y() + bar.get_height() / 2,
             f'{r2_models[i]:.5f}',
             fontsize=10, fontweight='bold', va='center', ha='left')

plt.title('R² Score of Models', fontsize=14, fontweight='bold')
plt.xlabel('R²', fontsize=12, fontweight='bold')
plt.ylabel('Model', fontsize=12, fontweight='bold')
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.xticks(fontsize=10, fontweight='bold')
plt.yticks(fontsize=10, fontweight='bold')
plt.tight_layout()
plt.show()
```

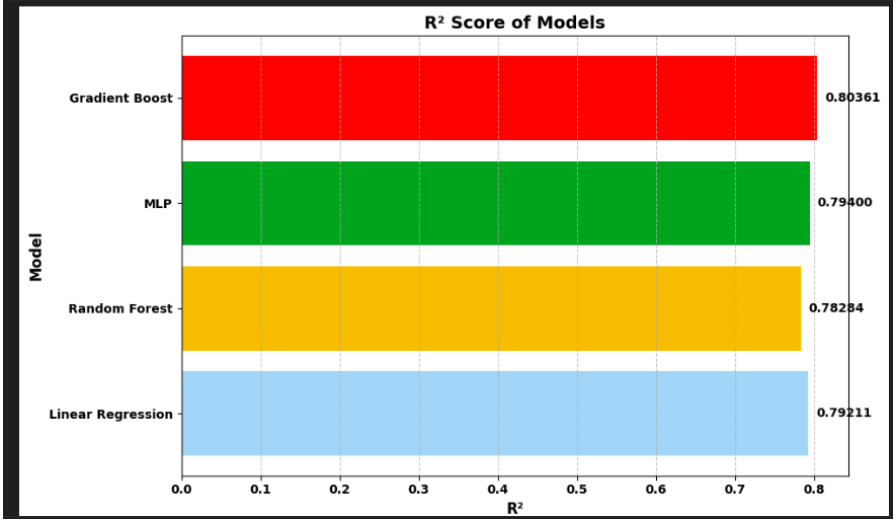


Figure 35: Bar Plot Showing R^2 Score for Each Model

7.2 Hyperparameter Tuning

Hyperparameter tuning was performed using GridSearchCV to try to improve the Gradient Boosting model. It tried all combinations of parameters defined, such as the number of estimators, maximum depth, and learning rate, on cross-validation to find the best combination that would give the lowest Mean Absolute Error. The best combination was found, and the model with the tuned parameters was trained and evaluated; it outperformed the model with the default settings. Below is a heat map visualizing how the different combinations of parameters affect the model performance.

The following are the codes and results of the hyperparameter tuning:

Hyperparameter Tuning

```
gb_param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
}

gb_model = GradientBoostingRegressor(random_state=42)

gb_grid_search = GridSearchCV(estimator=gb_model, param_grid=gb_param_grid, cv=10,
                              scoring='neg_mean_absolute_error', n_jobs=-1, verbose=2)

gb_grid_search.fit(X_train, y_train)

best_params = gb_grid_search.best_params_
best_score = -gb_grid_search.best_score_

print("Best Parameters for Gradient Boosting:", best_params)
print("Best Mean Absolute Error (MAE) from CV:", best_score)
```

```
Fitting 10 folds for each of 27 candidates, totalling 270 fits
Best Parameters for Gradient Boosting: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100}
Best Mean Absolute Error (MAE) from CV: 0.2519799229359044
```

Figure 36: Code for Hyperparameter Tuning with GridSearchCV

```
results = pd.DataFrame(gb_grid_search.cv_results_)

# Create a pivot table for heatmap
heatmap_data = results.pivot_table(
    index='param_max_depth',
    columns='param_learning_rate',
    values='mean_test_score'
)

# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(heatmap_data, annot=True, fmt=".4f", cmap="YlGnBu", cbar_kws={'label': 'Negative MAE'})
plt.title('Hyperparameter Tuning Results (Gradient Boosting)', fontsize=14, fontweight='bold')
plt.xlabel('Learning Rate', fontsize=12, fontweight='bold')
plt.ylabel('Max Depth', fontsize=12, fontweight='bold')
plt.xticks(fontsize=10, fontweight='bold')
plt.yticks(fontsize=10, fontweight='bold')
plt.tight_layout()
plt.show()
```

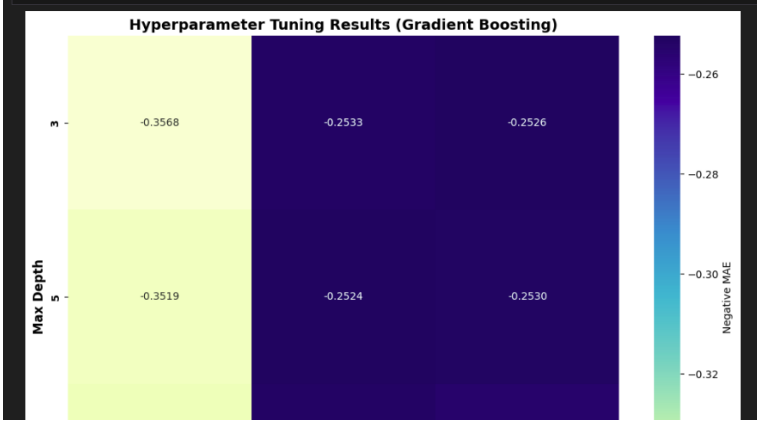


Figure 37: Heatmap of Hyperparameter Tuning Results

```
optimized_params = gb_grid_search.best_params_

gb_model_optimized = GradientBoostingRegressor(
    random_state=42,
    n_estimators=optimized_params['n_estimators'],
    max_depth=optimized_params['max_depth'],
    learning_rate=optimized_params['learning_rate']
)

gb_model_optimized.fit(X_train, y_train)

y_pred_gb_optimized = gb_model_optimized.predict(X_test)

mae_gb_optimized = mean_absolute_error(y_test, y_pred_gb_optimized)
rmse_gb_optimized = np.sqrt(mean_squared_error(y_test, y_pred_gb_optimized))
r2_gb_optimized = r2_score(y_test, y_pred_gb_optimized)

mae_gb_optimized, rmse_gb_optimized, r2_gb_optimized
```

```
(0.2543846426386597, 0.4407867486763877, 0.8039180047569985)
```

Figure 38: Evaluation of the Optimized Gradient Boosting Model

8 Testing and Deployment

For testing and deploying the model, a number of scenarios were taken to make it close to the real world. Further, functions were created to convert the input data to a corresponding scale for making predictions and then scaling back the same prediction to its original values. Different test cases were elaborated in order to understand model performance for different situations-like distance, number of passengers, peak and off-peak hours, weekdays or weekends, and seasons. The obtained results were analyzed in order to get some practical insight for applications in real life.

Function to transform input data to form of data on which model was trained.

```
def transform_scenario(hour, weekday, month, passenger_count, distance, scaler):
    input_for_scaler = pd.DataFrame({
        "fare_amount": [0],
        "ride_distance": [distance]
    })
    scaled_values = scaler.transform(input_for_scaler)
    scaled_distance = scaled_values[0][1]
    hour_sin = np.sin(2 * np.pi * hour / 24)
    hour_cos = np.cos(2 * np.pi * hour / 24)
    weekday_sin = np.sin(2 * np.pi * weekday / 7)
    weekday_cos = np.cos(2 * np.pi * weekday / 7)
    month_encoding = [False] * 12
    month_encoding[month - 1] = True
    features_dict = {
        "passenger_count": passenger_count,
        "ride_distance": scaled_distance,
        "Hour_sin": hour_sin,
        "Hour_cos": hour_cos,
        "WeekDay_sin": weekday_sin,
        "WeekDay_cos": weekday_cos,
    }
    for i, value in enumerate(month_encoding, start=1):
        features_dict[f"Month_{i}"] = value
    return pd.DataFrame([features_dict])
```

Figure 39: Function to transform input data for model prediction.

Function to inverse tranform fare prices

```python
def get_original_fare(predicted_scaled_fare, scaler):
    inverse_transformed = scaler.inverse_transform([[predicted_scaled_fare, 0]])
    original_fare = inverse_transformed[0][0]
    return max(0, original_fare)
```

Figure 40: Function to inverse-transform fare predictions to original scale.

First Scenario

```python
distance = 10
hour = 8
weekday = 0
month = 4

# Scenario A: 1 passenger
passenger_count_a = 1
input_features_a = transform_scenario(hour, weekday, month, passenger_count_a, distance, scaler)
fare_prediction_a_scaled = gb_model_optimized.predict(input_features_a)[0]

# Convert scaled fare to original fare
fare_prediction_a = get_original_fare(fare_prediction_a_scaled, scaler)
print(f"Predicted Fare for Scenario A (1 passenger): ${fare_prediction_a:.2f}")

# Scenario B: 4 passengers
passenger_count_b = 4
input_features_b = transform_scenario(hour, weekday, month, passenger_count_b, distance, scaler)
fare_prediction_b_scaled = gb_model_optimized.predict(input_features_b)[0]

# Convert scaled fare to original fare
fare_prediction_b = get_original_fare(fare_prediction_b_scaled, scaler)
print(f"Predicted Fare for Scenario B (4 passengers): ${fare_prediction_b:.2f}")
```

```
Predicted Fare for Scenario A (1 passenger): $26.79
Predicted Fare for Scenario B (4 passengers): $27.15
```

Figure 41: First scenario testing fare for one and four passengers for a specific distance and time.

Second Scenario

```python
distance = 5
weekday = 0
month = 1
passenger_count = 2

# Off-Peak Morning Ride
hour_off_peak = 10
input_features_off_peak = transform_scenario(
    hour=hour_off_peak,
    weekday=weekday,
    month=month,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_off_peak_scaled = gb_model_optimized.predict(input_features_off_peak)[0]

# Convert scaled fare to original fare
fare_prediction_off_peak = get_original_fare(fare_prediction_off_peak_scaled, scaler)
print(f"Predicted Fare for Off-Peak Morning Ride (10:00 AM): ${fare_prediction_off_peak:.2f}")

# Peak Evening Ride
hour_peak = 18  # 6:00 PM
input_features_peak = transform_scenario(
    hour=hour_peak,
    weekday=weekday,
    month=month,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_peak_scaled = gb_model_optimized.predict(input_features_peak)[0]

# Convert scaled fare to original fare
fare_prediction_peak = get_original_fare(fare_prediction_peak_scaled, scaler)
print(f"Predicted Fare for Peak Evening Ride (6:00 PM): ${fare_prediction_peak:.2f}")
```

```
Predicted Fare for Off-Peak Morning Ride (10:00 AM): $12.20
Predicted Fare for Peak Evening Ride (6:00 PM): $11.04
```

Figure 42: Second scenario testing peak and off-peak fares for the same ride.

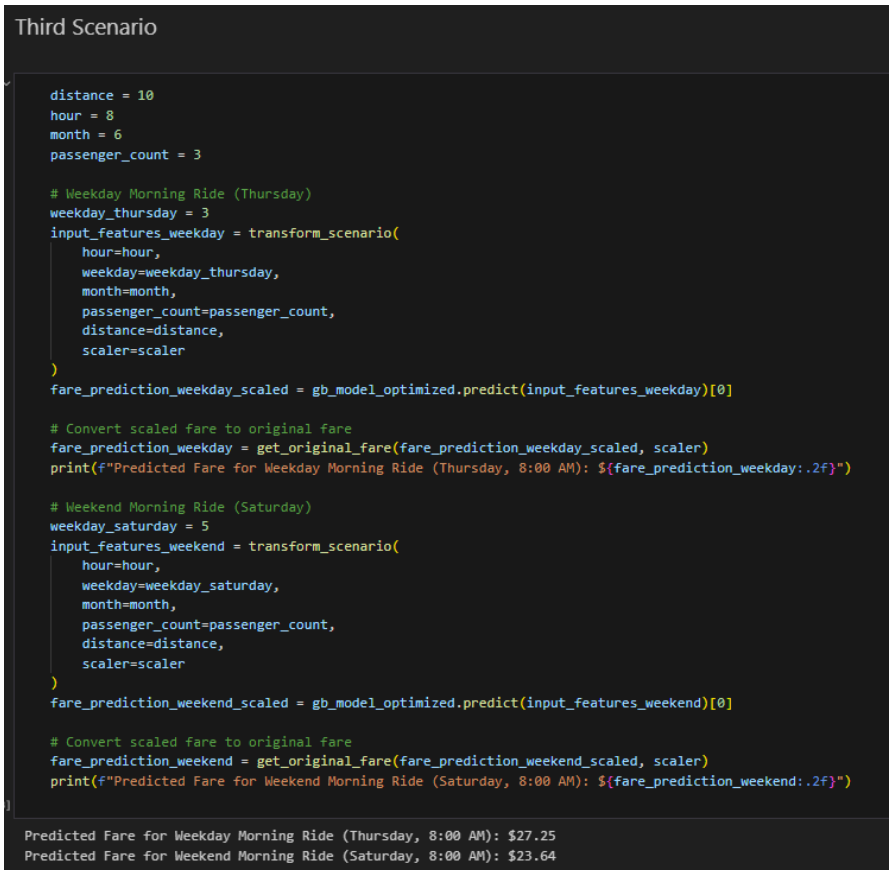


Figure 43: Third scenario testing weekday and weekend morning ride fares.

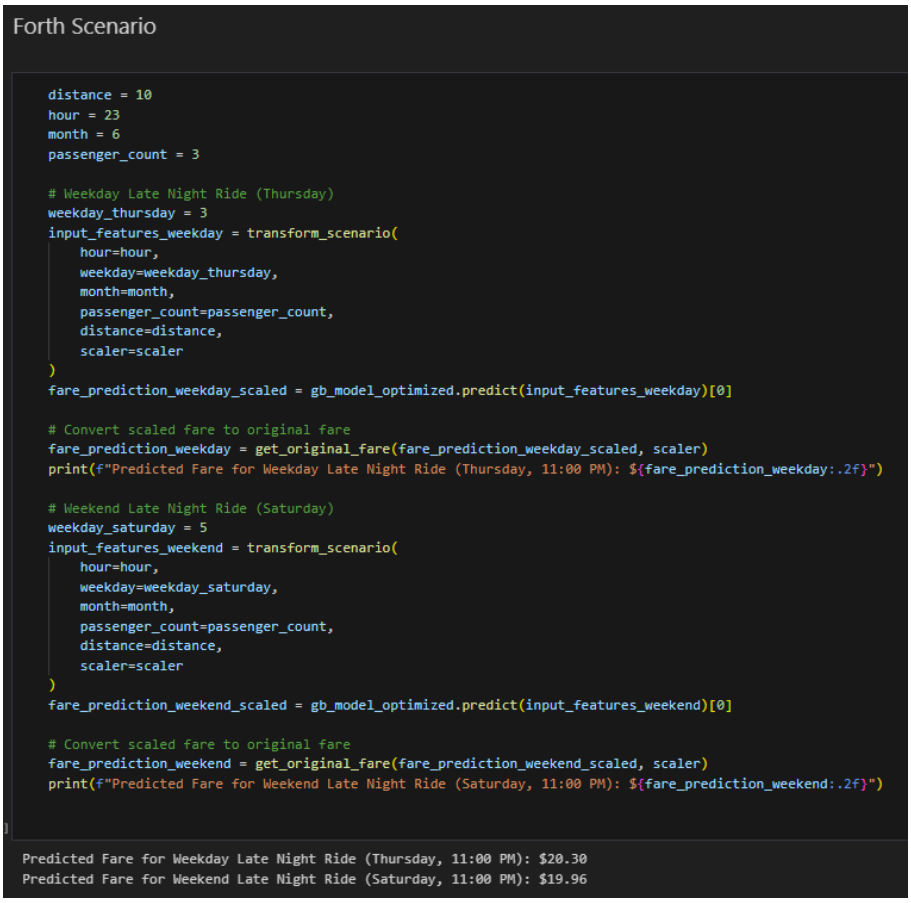


Figure 44: Fourth scenario testing weekday and weekend late-night ride fares.

Fifth Scenario

```
distance = 7.5
hour = 10
weekday = 0
passenger_count = 1

# Spring Ride (April)
month_april = 4
input_features_spring = transform_scenario(
    hour=hour,
    weekday=weekday,
    month=month_april,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_spring_scaled = gb_model_optimized.predict(input_features_spring)[0]

# Convert scaled fare to original fare
fare_prediction_spring = get_original_fare(fare_prediction_spring_scaled, scaler)
print(f"Predicted Fare for Spring Ride (April, 10:00 AM): ${fare_prediction_spring:.2f}")

# Holiday Season Ride (December)
month_december = 12
input_features_holiday = transform_scenario(
    hour=hour,
    weekday=weekday,
    month=month_december,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_holiday_scaled = gb_model_optimized.predict(input_features_holiday)[0]

# Convert scaled fare to original fare
fare_prediction_holiday = get_original_fare(fare_prediction_holiday_scaled, scaler)
print(f"Predicted Fare for Holiday Season Ride (December, 10:00 AM): ${fare_prediction_holiday:.2f}")
```

```
Predicted Fare for Spring Ride (April, 10:00 AM): $17.69
Predicted Fare for Holiday Season Ride (December, 10:00 AM): $18.36
```

Figure 45: Fifth scenario testing spring and holiday season ride fares.

Sixth Scenario

```
distance = 15
weekday = 5
month = 2
passenger_count = 3

# Late-Night Ride at 1:00 AM
hour_1am = 1
input_features_1am = transform_scenario(
    hour=hour_1am,
    weekday=weekday,
    month=month,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_1am_scaled = gb_model_optimized.predict(input_features_1am)[0]

# Convert scaled fare to original fare
fare_prediction_1am = get_original_fare(fare_prediction_1am_scaled, scaler)
print(f"Predicted Fare for Late-Night Ride (Saturday, 1:00 AM): ${fare_prediction_1am:.2f}")

# Late-Night Ride at 3:00 AM
hour_3am = 3
input_features_3am = transform_scenario(
    hour=hour_3am,
    weekday=weekday,
    month=month,
    passenger_count=passenger_count,
    distance=distance,
    scaler=scaler
)
fare_prediction_3am_scaled = gb_model_optimized.predict(input_features_3am)[0]

# Convert scaled fare to original fare
fare_prediction_3am = get_original_fare(fare_prediction_3am_scaled, scaler)
print(f"Predicted Fare for Late-Night Ride (Saturday, 3:00 AM): ${fare_prediction_3am:.2f}")
```

```
Predicted Fare for Late-Night Ride (Saturday, 1:00 AM): $29.94
Predicted Fare for Late-Night Ride (Saturday, 3:00 AM): $30.98
```

Figure 46: Sixth scenario testing late-night ride fares at different times.