Uber Trip Time Series Analysis

Author:

Suman Senapati

Overview

This comprehensive dataset provides detailed information about ride-hailing services in New York City, with a primary focus on Uber operations during 2014-2015. The data was obtained through Freedom of Information Law (FOIL) requests submitted by FiveThirtyEight to the NYC Taxi & Limousine Commission (TLC) in July 2015.

Dataset Source:

source: https://github.com/fivethirtyeight/uber-tlc-foil-response

Dataset Scope

- **Uber Pickups**: Over 18.8 million records
  - 4.5+ million pickups (April-September 2014)
  - 14.3+ million pickups (January-June 2015)
- **Additional FHV Data**:
  - Individual trip data from 10 for-hire vehicle companies
  - Aggregated data covering 329 FHV companies

Research Impact

The dataset has been instrumental in several significant FiveThirtyEight investigations:

1. Analysis of Uber's service coverage in NYC's outer boroughs compared to traditional taxis
2. Examination of the relationship between public transit and ride-hailing services
3. Assessment of Uber's impact on Manhattan's traditional taxi market
4. Investigation into Uber's effect on NYC rush-hour traffic patterns

Data Structure

2014 Uber Data (April-September)

Files are organized monthly with the following columns:

- Date/Time: Pickup timestamp
- Lat: Pickup latitude
- Lon: Pickup longitude
- Base: TLC base company code

2015 Uber Data (January-June)

Contains enhanced data fields:

- Dispatching_base_num: TLC base company dispatch code
- Pickup_date: Pickup timestamp
- Affiliated_base_num: TLC base company affiliation code
- locationID: Standardized pickup location identifier

Base Code Reference

Uber operates through multiple bases in NYC, each with a unique identifier:

- B02512: Unter
- B02598: Hinter
- B02617: Weiter
- B02682: Schmecken
- B02764: Danach-NY
- B02765: Grun
- B02835: Dreist
- B02836: Drinnen

Supplementary Data

FHV Company Data

The dataset includes:

- Detailed trip records from 10 FHV companies
- Daily pickup aggregates for 329 FHV companies (January-August 2015)
- Variable fields including:
  - Trip date and time
  - Pickup locations
  - Driver license numbers
  - Vehicle license numbers

Reference Files

- taxi-zone-lookup.csv: Maps locationIDs to specific zones and boroughs
- Aggregate_FHV_Data.xlsx: TLC's analysis of taxi and FHV trips
- Uber-Jan-Feb-FOIL.csv: Daily aggregated Uber statistics (January-February 2015)
- TLC correspondence: Documentation of the FOIL request process (TLC_letter.pdf series)

Data Collection Context

The data was released in multiple batches throughout 2015 as the TLC reviewed and processed the records. This systematic release ensures comprehensive coverage while maintaining data quality standards. The inclusion of TLC correspondence provides transparency regarding the data collection process and any potential limitations or considerations in the dataset.

Technical Notes

1. Location data is provided in two formats:
   - Direct geographical coordinates (2014 data)
   - LocationID reference system (2015 data)
2. Base codes serve as important identifiers for tracking operational patterns
3. The dataset structure evolved from 2014 to 2015, reflecting enhanced data collection practices

This dataset represents a significant resource for understanding the evolution of ride-hailing services in New York City and their impact on urban transportation patterns during a critical growth period.

Based on the dataset we are working on, we may look at the following goals/objectives:

1. Ride Demand Prediction

Goal: Predict the number of Uber pickups at a given time and location.

- **Input Features:**
  - Date & Time (hour, day, month, weekday/weekend)
  - Location (Latitude & Longitude or Zone ID)
- **Output:** Expected number of Uber pickups.
- **Models:**
  - **Time Series Models:** ARIMA, SARIMA, Facebook Prophet
  - **Machine Learning Models:** XGBoost, Random Forest, LSTM (Deep Learning)

Use Case: Helps Uber optimize driver deployment and dynamic pricing.

2. Trip Hotspot Prediction

Goal: Predict the most popular pickup locations at different times of the day.

- **Input Features:**
  - Time of Day, Day of Week
  - Past demand in the same location
- **Output:** Top **N** locations where demand will be high.
- **Models:**
  - **Clustering:** K-Means, DBSCAN (to identify ride hotspots)
  - **Classification Model:** Random Forest
  - **Time series:** LSTM

Use Case: Uber can allocate more drivers in high-demand areas.

3. Peak Hour Traffic Analysis

Goal: Predict when ride demand will surge based on past trends.

- **Input Features:**
  - Hourly Uber pickup trends
  - Day of the week, Holiday indicators
  - External data (weather, events)
- **Output:** Predict peak ride demand hours.
- **Models:**
  - Time Series Forecasting (LSTM, ARIMA, Prophet)
  - Classification (Random Forest, XGBoost)

Use Case: Helps Uber optimize surge pricing and driver incentives.

---

0. Library Installation (If Any)

Some libraries are additionally needed for the analysis and model development

In [1]:
```
pip install folium geopandas contextily geoplot
```

```
Requirement already satisfied: folium in /usr/local/lib/python3.10/dist-packages (0.
19.2)
Requirement already satisfied: geopandas in /usr/local/lib/python3.10/dist-packages
(0.14.4)
Collecting contextily
  Downloading contextily-1.6.2-py3-none-any.whl.metadata (2.9 kB)
Collecting geoplot
  Downloading geoplot-0.5.1-py3-none-any.whl.metadata (1.7 kB)
Requirement already satisfied: branca>=0.6.0 in /usr/local/lib/python3.10/dist-packa
ges (from folium) (0.8.1)
Requirement already satisfied: jinja2>=2.9 in /usr/local/lib/python3.10/dist-package
s (from folium) (3.1.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (fro
m folium) (1.26.4)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages
(from folium) (2.32.3)
Requirement already satisfied: xyzservices in /usr/local/lib/python3.10/dist-package
s (from folium) (2024.9.0)
Requirement already satisfied: fiona>=1.8.21 in /usr/local/lib/python3.10/dist-packa
ges (from geopandas) (1.10.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages
(from geopandas) (24.2)
Requirement already satisfied: pandas>=1.4.0 in /usr/local/lib/python3.10/dist-packa
ges (from geopandas) (2.2.2)
Requirement already satisfied: pyproj>=3.3.0 in /usr/local/lib/python3.10/dist-packa
ges (from geopandas) (3.7.0)
Requirement already satisfied: shapely>=1.8.0 in /usr/local/lib/python3.10/dist-pack
ages (from geopandas) (2.0.6)
Requirement already satisfied: geopy in /usr/local/lib/python3.10/dist-packages (fro
m contextily) (2.4.1)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages
(from contextily) (3.7.5)
Collecting mercantile (from contextily)
  Downloading mercantile-1.2.1-py3-none-any.whl.metadata (4.8 kB)
Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages (fr
om contextily) (11.0.0)
Collecting rasterio (from contextily)
  Downloading rasterio-1.4.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.
whl.metadata (9.1 kB)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages (fr
om contextily) (1.4.2)
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (f
rom geoplot) (0.12.2)
Requirement already satisfied: cartopy in /usr/local/lib/python3.10/dist-packages (f
rom geoplot) (0.24.1)
Collecting mapclassify>=2.1 (from geoplot)
  Downloading mapclassify-2.8.1-py3-none-any.whl.metadata (2.8 kB)
Requirement already satisfied: attrs>=19.2.0 in /usr/local/lib/python3.10/dist-packa
ges (from fiona>=1.8.21->geopandas) (24.3.0)
Requirement already satisfied: certifi in /usr/local/lib/python3.10/dist-packages (f
rom fiona>=1.8.21->geopandas) (2024.12.14)
Requirement already satisfied: click~=8.0 in /usr/local/lib/python3.10/dist-packages
(from fiona>=1.8.21->geopandas) (8.1.7)
Requirement already satisfied: click-plugins>=1.0 in /usr/local/lib/python3.10/dist-
packages (from fiona>=1.8.21->geopandas) (1.1.1)
Requirement already satisfied: cligj>=0.5 in /usr/local/lib/python3.10/dist-packages
```

```
(from fiona>=1.8.21->geopandas) (0.7.2)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-pac
kages (from jinja2>=2.9->folium) (3.0.2)
Requirement already satisfied: networkx>=2.7 in /usr/local/lib/python3.10/dist-packa
ges (from mapclassify>=2.1->geoplot) (3.4.2)
Requirement already satisfied: scikit-learn>=1.0 in /usr/local/lib/python3.10/dist-p
ackages (from mapclassify>=2.1->geoplot) (1.2.2)
Requirement already satisfied: scipy>=1.8 in /usr/local/lib/python3.10/dist-packages
(from mapclassify>=2.1->geoplot) (1.13.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-pa
ckages (from matplotlib->contextily) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packag
es (from matplotlib->contextily) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-p
ackages (from matplotlib->contextily) (4.55.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-p
ackages (from matplotlib->contextily) (1.4.7)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-pa
ckages (from matplotlib->contextily) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dis
t-packages (from matplotlib->contextily) (2.8.2)
Requirement already satisfied: mkl_fft in /usr/local/lib/python3.10/dist-packages (f
rom numpy->folium) (1.3.8)
Requirement already satisfied: mkl_random in /usr/local/lib/python3.10/dist-packages
(from numpy->folium) (1.2.4)
Requirement already satisfied: mkl_umath in /usr/local/lib/python3.10/dist-packages
(from numpy->folium) (0.1.1)
Requirement already satisfied: mkl in /usr/local/lib/python3.10/dist-packages (from
numpy->folium) (2025.0.1)
Requirement already satisfied: tbb4py in /usr/local/lib/python3.10/dist-packages (fr
om numpy->folium) (2022.0.0)
Requirement already satisfied: mkl-service in /usr/local/lib/python3.10/dist-package
s (from numpy->folium) (2.4.1)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packag
es (from pandas>=1.4.0->geopandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-pack
ages (from pandas>=1.4.0->geopandas) (2024.2)
Requirement already satisfied: pyshp>=2.3 in /usr/local/lib/python3.10/dist-packages
(from cartopy->geoplot) (2.3.1)
Requirement already satisfied: geographiclib<3,>=1.52 in /usr/local/lib/python3.10/d
ist-packages (from geopy->contextily) (2.0)
Collecting affine (from rasterio->contextily)
  Downloading affine-2.4.0-py3-none-any.whl.metadata (4.0 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.1
0/dist-packages (from requests->folium) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packag
es (from requests->folium) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-
packages (from requests->folium) (2.2.3)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages
(from python-dateutil>=2.7->matplotlib->contextily) (1.17.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dis
t-packages (from scikit-learn>=1.0->mapclassify>=2.1->geoplot) (3.5.0)
Requirement already satisfied: intel-openmp>=2024 in /usr/local/lib/python3.10/dist-
packages (from mkl->numpy->folium) (2024.2.0)
Requirement already satisfied: tbb==2022.* in /usr/local/lib/python3.10/dist-package
```

```
s (from mkl->numpy->folium) (2022.0.0)
Requirement already satisfied: tcmlib==1.* in /usr/local/lib/python3.10/dist-package
s (from tbb==2022.*->mkl->numpy->folium) (1.2.0)
Requirement already satisfied: intel-cmplr-lib-rt in /usr/local/lib/python3.10/dist-
packages (from mkl_umath->numpy->folium) (2024.2.0)
Requirement already satisfied: intel-cmplr-lib-ur==2024.2.0 in /usr/local/lib/python
3.10/dist-packages (from intel-openmp>=2024->mkl->numpy->folium) (2024.2.0)
Downloading contextily-1.6.2-py3-none-any.whl (17 kB)
Downloading geoplot-0.5.1-py3-none-any.whl (28 kB)
Downloading mapclassify-2.8.1-py3-none-any.whl (59 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 59.1/59.1 kB 3.4 MB/s eta 0:00:00
Downloading mercantile-1.2.1-py3-none-any.whl (14 kB)
Downloading rasterio-1.4.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.wh
l (22.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 22.2/22.2 MB 61.4 MB/s eta 0:00:00
Downloading affine-2.4.0-py3-none-any.whl (15 kB)
Installing collected packages: mercantile, affine, rasterio, mapclassify, contextil
y, geoplot
Successfully installed affine-2.4.0 contextily-1.6.2 geoplot-0.5.1 mapclassify-2.8.1
mercantile-1.2.1 rasterio-1.4.3
Note: you may need to restart the kernel to use updated packages.
```

1. Importing Libraries

The following python libraries are used throughout the notebook for analysis, model development and evaluation

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import glob
import folium
from folium.plugins import HeatMap
import geopandas
import kagglehub
from kagglehub import KaggleDatasetAdapter
import geoplot
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from sklearn.cluster import KMeans, DBSCAN
from sklearn.mixture import GaussianMixture
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
from xgboost import XGBRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.metrics import silhouette_score, davies_bouldin_score, calinski_haraba
import joblib
from datetime import datetime, timedelta
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

2. Importing datasets

The dataset we are working on has separate files on uber rides data, with various dates. We shall target to use data from these files and use the sets which have complete information.

Dataset 1: Pickup date and base number

In [3]:
```
pickup_df = pd.read_csv("/kaggle/input/uber-trip-data-jan-june-2015/uber-raw-data-j
```

Dataset 2: Number of Trip, Active vehicles and base

In [4]:
```
trip_veh_df = pd.read_csv("/kaggle/input/uber-trip-data-jan-and-feb-2015/Uber-Jan-F
```

Dataset 3:

In [5]:
```
csv_files = glob.glob("/kaggle/input/uber-raw-data-apr-2014-sep-2014/*.csv")
dfs = [pd.read_csv(file) for file in csv_files]
lat_long_df = pd.concat(dfs, ignore_index=True)
```

3. Data Pre-processing

In [6]:
```
pickup_df.head()
```

Out[6]:

	Dispatching_base_num	Pickup_date	Affiliated_base_num	locationID
0	B02617	2015-05-17 09:47:00	B02617	141
1	B02617	2015-05-17 09:47:00	B02617	65
2	B02617	2015-05-17 09:47:00	B02617	100
3	B02617	2015-05-17 09:47:00	B02774	80
4	B02617	2015-05-17 09:47:00	B02617	90

In [7]:
```
trip_veh_df.head()
```

Out[7]:

	dispatching_base_number	date	active_vehicles	trips
0	B02512	1/1/2015	190	1132
1	B02765	1/1/2015	225	1765
2	B02764	1/1/2015	3427	29421
3	B02682	1/1/2015	945	7679
4	B02617	1/1/2015	1228	9537

In [8]:
```
lat_long_df.head()
```

Out[8]:

	Date/Time	Lat	Lon	Base
0	5/1/2014 0:02:00	40.7521	-73.9914	B02512
1	5/1/2014 0:06:00	40.6965	-73.9715	B02512
2	5/1/2014 0:15:00	40.7464	-73.9838	B02512
3	5/1/2014 0:17:00	40.7463	-74.0011	B02512
4	5/1/2014 0:17:00	40.7594	-73.9734	B02512

So, now we have three dataframes consisting of the data from different input files. We need to sensibly merge them before proceeding

In [9]:
```
pickup_df.shape
```

Out[9]:
```
(14270479, 4)
```

In [10]:
```
trip_veh_df.shape
```

Out[10]:
```
(354, 4)
```

In [11]:
```
lat_long_df.shape
```

Out[11]:
```
(4534327, 4)
```

As can be seen from the dataframe shapes that the vehicle and trip data is pretty scarce. Let's explore the range of dates which is covered in these

In [12]:
```
pickup_df["DateTime"] = pd.to_datetime(pickup_df["Pickup_date"])
trip_veh_df["DateTime"] = pd.to_datetime(trip_veh_df["date"])
lat_long_df["DateTime"] = pd.to_datetime(lat_long_df["Date/Time"])
```

In [13]:
```
pickup_df.dtypes
```

Out[13]:
```
Dispatching_base_num            object
Pickup_date                     object
Affiliated_base_num             object
locationID                       int64
DateTime                datetime64[ns]
dtype: object
```

In [14]:
```
trip_veh_df.dtypes
```

Out[14]:
```
dispatching_base_number            object
date                               object
active_vehicles                     int64
trips                               int64
DateTime                   datetime64[ns]
dtype: object
```

In [15]:
```
lat_long_df.dtypes
```

Out[15]:
```
Date/Time            object
Lat                 float64
Lon                 float64
Base                 object
DateTime     datetime64[ns]
dtype: object
```

In [16]:
```
base_mapping = {
    "B02512": "Unter",
    "B02598": "Hinter",
    "B02617": "Weiter",
    "B02682": "Schmecken",
    "B02764": "Danach-NY",
    "B02765": "Grun",
    "B02835": "Dreist",
    "B02836": "Drinnen"
}
```

In [17]:
```
pickup_df['Base Name'] = pickup_df['Dispatching_base_num'].map(base_mapping)
trip_veh_df['Base Name'] = trip_veh_df['dispatching_base_number'].map(base_mapping)
lat_long_df['Base Name'] = lat_long_df['Base'].map(base_mapping)
```

The oldest and newest entries in the dataframes are checked to understand how the dataframes can be merged or used together

In [18]:
```
print(f"Oldest date in dataframe 1 is {pickup_df['DateTime'].min()} and newest date
print(f"Oldest date in dataframe 2 is {trip_veh_df['DateTime'].min()} and newest da
print(f"Oldest date in dataframe 3 is {lat_long_df['DateTime'].min()} and newest da
```

```
Oldest date in dataframe 1 is 2015-01-01 00:00:05 and newest date is 2015-06-30 23:5
9:00
Oldest date in dataframe 2 is 2015-01-01 00:00:00 and newest date is 2015-02-28 00:0
0:00
Oldest date in dataframe 3 is 2014-04-01 00:00:00 and newest date is 2014-09-30 22:5
9:00
```

Dataframe 3 cannot be used alongside the other two dataframes. In this notebook, we will explore the trip hotspots using dataset 3, and try to predict the top hotspots based on date and time

In [19]:
```
print(lat_long_df.head(5))
```

```
         Date/Time      Lat      Lon    Base            DateTime Base Name
0  5/1/2014 0:02:00  40.7521 -73.9914  B02512 2014-05-01 00:02:00     Unter
1  5/1/2014 0:06:00  40.6965 -73.9715  B02512 2014-05-01 00:06:00     Unter
2  5/1/2014 0:15:00  40.7464 -73.9838  B02512 2014-05-01 00:15:00     Unter
3  5/1/2014 0:17:00  40.7463 -74.0011  B02512 2014-05-01 00:17:00     Unter
4  5/1/2014 0:17:00  40.7594 -73.9734  B02512 2014-05-01 00:17:00     Unter
```

For simplicity, we will copy the lat_long_df to a new dataframe named 'df'

In [20]:
```
df = lat_long_df.copy(deep=True)
```

In [21]:
```
df['Year'] = df['DateTime'].dt.year
df['Month'] = df['DateTime'].dt.month
df['Day'] = df['DateTime'].dt.day
df['Hour'] = df['DateTime'].dt.hour
df['Minute'] = df['DateTime'].dt.minute
df['DayOfWeek'] = df['DateTime'].dt.dayofweek
```

In [22]:
```
df.isna().sum().sum()
```

Out[22]:
```
0
```

In [23]:
```
print(df.info())
print(df.describe())
print(df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4534327 entries, 0 to 4534326
Data columns (total 12 columns):
 #   Column     Dtype
---  ------     -----
 0   Date/Time  object
 1   Lat        float64
 2   Lon        float64
 3   Base       object
 4   DateTime   datetime64[ns]
 5   Base Name  object
 6   Year       int32
 7   Month      int32
 8   Day        int32
 9   Hour       int32
 10  Minute     int32
 11  DayOfWeek  int32
dtypes: datetime64[ns](1), float64(2), int32(6), object(3)
memory usage: 311.3+ MB
None
```

	Lat	Lon	DateTime	Year	\
count	4.534327e+06	4.534327e+06	4534327	4534327.0	
mean	4.073926e+01	-7.397302e+01	2014-07-11 18:50:50.578151936	2014.0	
min	3.965690e+01	-7.492900e+01	2014-04-01 00:00:00	2014.0	
25%	4.072110e+01	-7.399650e+01	2014-05-28 15:18:00	2014.0	
50%	4.074220e+01	-7.398340e+01	2014-07-17 14:45:00	2014.0	
75%	4.076100e+01	-7.396530e+01	2014-08-27 21:55:00	2014.0	
max	4.211660e+01	-7.206660e+01	2014-09-30 22:59:00	2014.0	
std	3.994991e-02	5.726670e-02	NaN	0.0	

	Month	Day	Hour	Minute	DayOfWeek
count	4.534327e+06	4.534327e+06	4.534327e+06	4.534327e+06	4.534327e+06
mean	6.828703e+00	1.594337e+01	1.421831e+01	2.940071e+01	2.968115e+00
min	4.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	5.000000e+00	9.000000e+00	1.000000e+01	1.400000e+01	1.000000e+00
50%	7.000000e+00	1.600000e+01	1.500000e+01	2.900000e+01	3.000000e+00
75%	8.000000e+00	2.300000e+01	1.900000e+01	4.400000e+01	5.000000e+00
max	9.000000e+00	3.100000e+01	2.300000e+01	5.900000e+01	6.000000e+00
std	1.703810e+00	8.744902e+00	5.958759e+00	1.732238e+01	1.875971e+00

	Date/Time	Lat	Lon	Base	DateTime	Base Name	\
0	5/1/2014 0:02:00	40.7521	-73.9914	B02512	2014-05-01 00:02:00	Unter	
1	5/1/2014 0:06:00	40.6965	-73.9715	B02512	2014-05-01 00:06:00	Unter	
2	5/1/2014 0:15:00	40.7464	-73.9838	B02512	2014-05-01 00:15:00	Unter	
3	5/1/2014 0:17:00	40.7463	-74.0011	B02512	2014-05-01 00:17:00	Unter	
4	5/1/2014 0:17:00	40.7594	-73.9734	B02512	2014-05-01 00:17:00	Unter	

	Year	Month	Day	Hour	Minute	DayOfWeek
0	2014	5	1	0	2	3
1	2014	5	1	0	6	3
2	2014	5	1	0	15	3
3	2014	5	1	0	17	3
4	2014	5	1	0	17	3

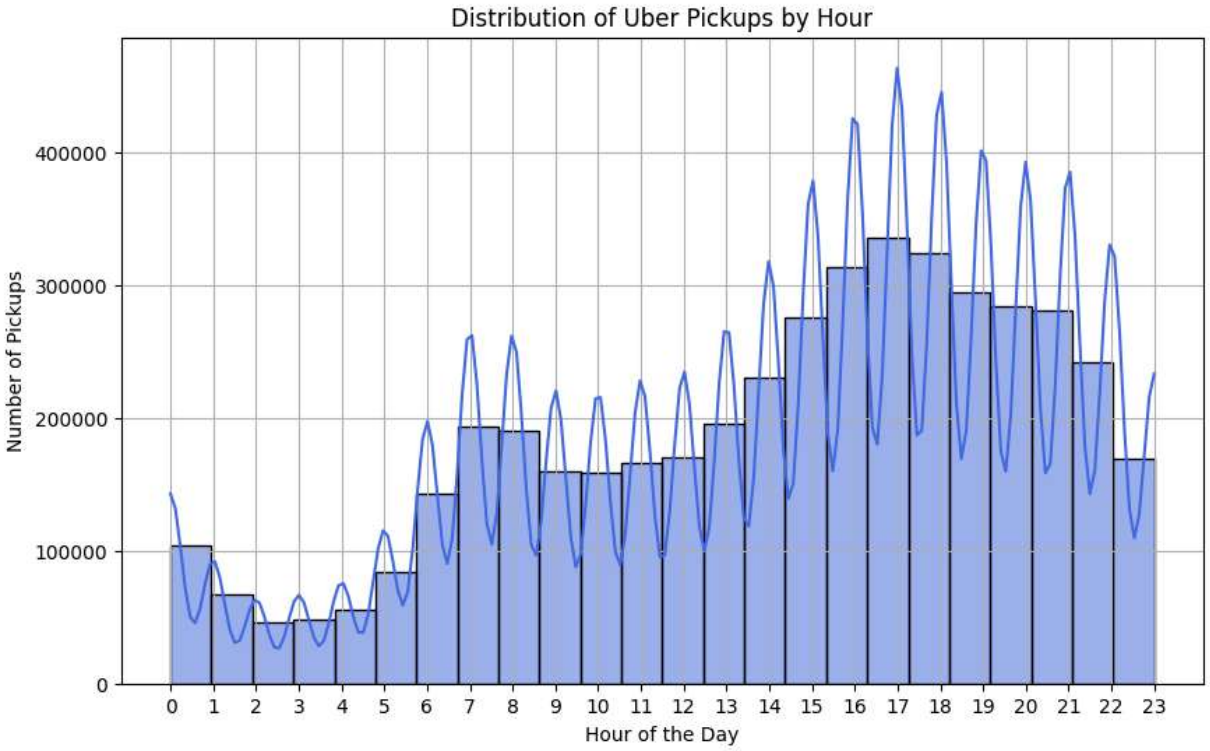
4. Exploratory Data Analysis

4.1 Univariate Analysis

Distribution of Pickups Over Time

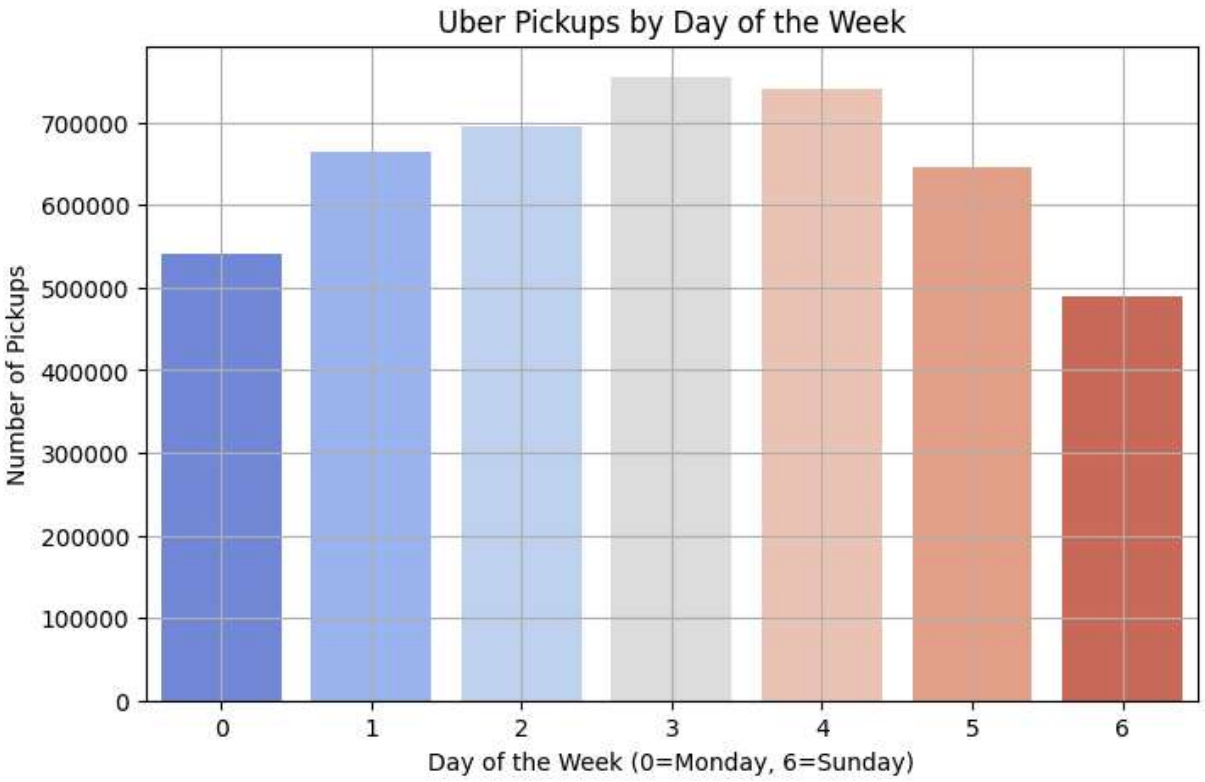
In [24]:
```
plt.figure(figsize=(10, 6))
sns.histplot(df['Hour'], bins=24, kde=True, color="royalblue")
plt.title("Distribution of Uber Pickups by Hour")
plt.xlabel("Hour of the Day")
plt.ylabel("Number of Pickups")
plt.xticks(range(24))
plt.grid()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



Distribution of Pickups by Day of the Week

In [25]:
```
plt.figure(figsize=(8, 5))
sns.countplot(x=df["DayOfWeek"], palette="coolwarm")
plt.title("Uber Pickups by Day of the Week")
plt.xlabel("Day of the Week (0=Monday, 6=Sunday)")
plt.ylabel("Number of Pickups")
plt.grid()
plt.show()
```



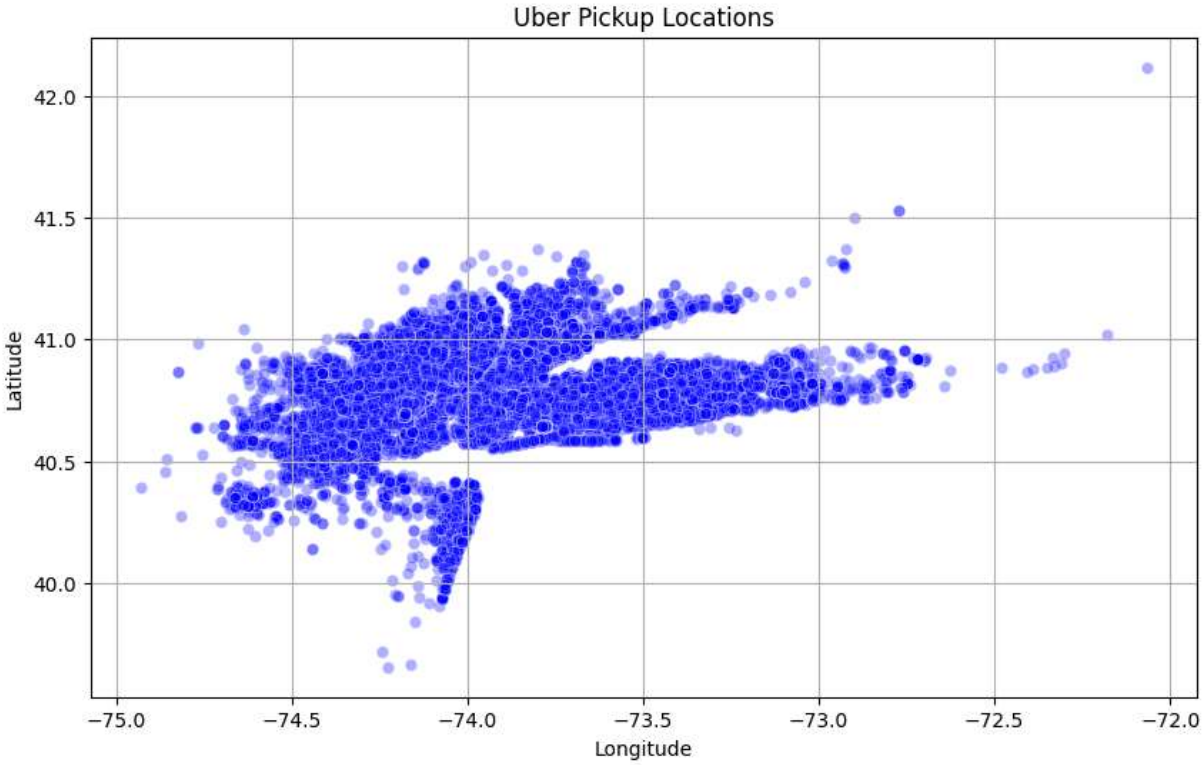
Base Distribution

In [26]:
```
plt.figure(figsize=(8, 5))
sns.countplot(x=df["Base Name"], palette="muted")
plt.title("Distribution of Uber Pickups by Base")
plt.xlabel("Base Name")
plt.ylabel("Number of Pickups")
plt.xticks(rotation=45)
plt.grid()
plt.show()
```

Geographic Distribution of Pickups

Using Scatterplot

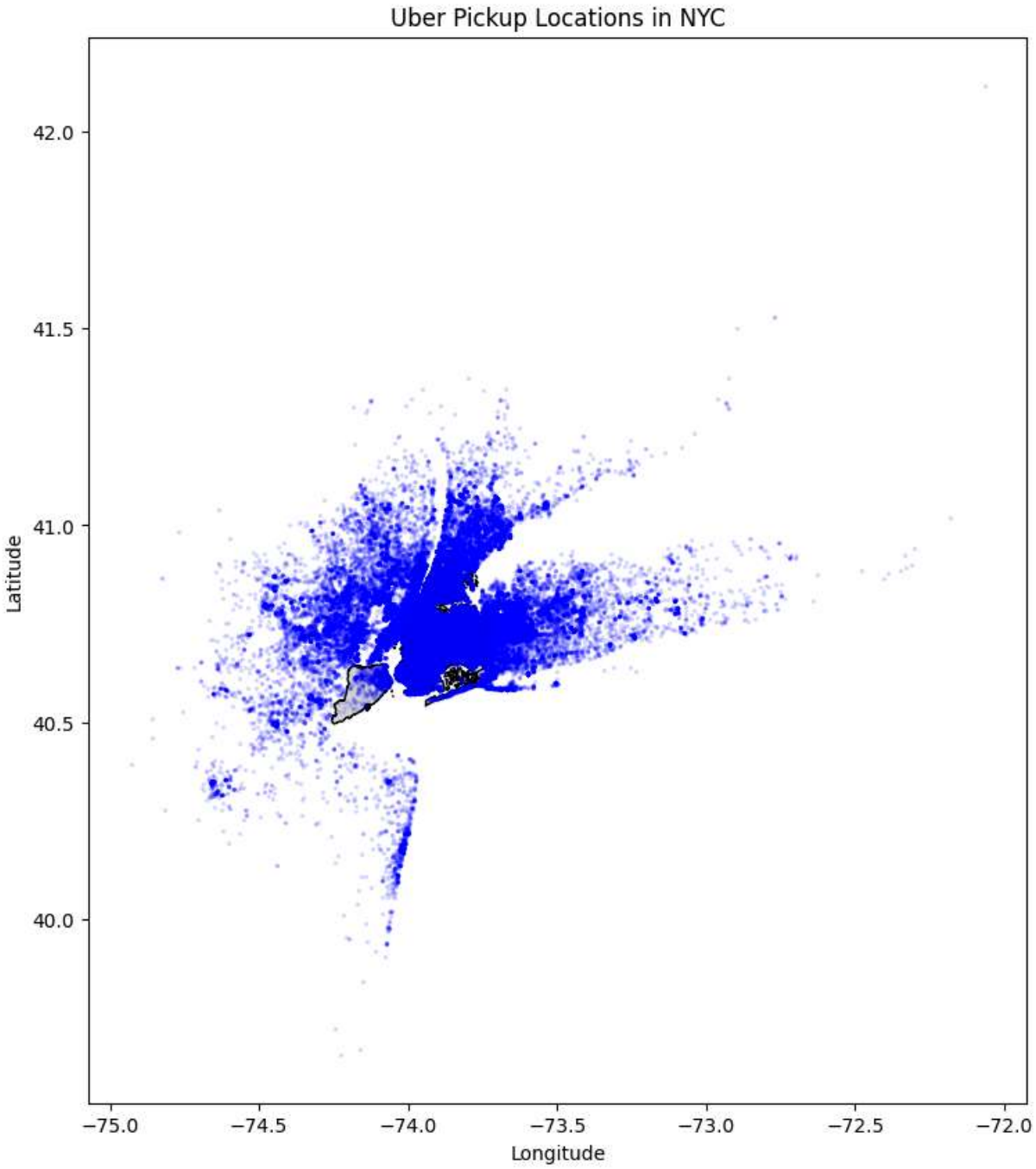
In [27]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x=df["Lon"], y=df["Lat"], alpha=0.3, color='blue')
plt.title("Uber Pickup Locations")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.grid()
plt.show()
```



Using Maps

In [28]:
```
nyc_map = geopandas.read_file("/kaggle/input/nyc-borough-boundaries/nybb.shp")

print("NYC Shapefile CRS:", nyc_map.crs)

crs={'init':'epsg:4326'}

boroughs = geopandas.read_file(geoplot.datasets.get_path('nyc_boroughs'))

gdf=geopandas.GeoDataFrame(df,crs=crs,geometry=geopandas.points_from_xy(df["Lon"],



fig, ax = plt.subplots(figsize=(10, 10))
boroughs.plot(ax=ax, color="lightgrey", edgecolor="black")
gdf.plot(ax=ax, markersize=2, color="blue", alpha=0.1)

plt.title("Uber Pickup Locations in NYC")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.show()
```

```
NYC Shapefile CRS: EPSG:2263
/usr/local/lib/python3.10/dist-packages/pyproj/crs/crs.py:143: FutureWarning: '+init
=<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred ini
tialization method. When making the change, be mindful of axis order changes: http
s://pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  in_crs_string = _prepare_from_proj_string(in_crs_string)
```



4.2 Multivariate Analysis

4.2.1 Pair Plot

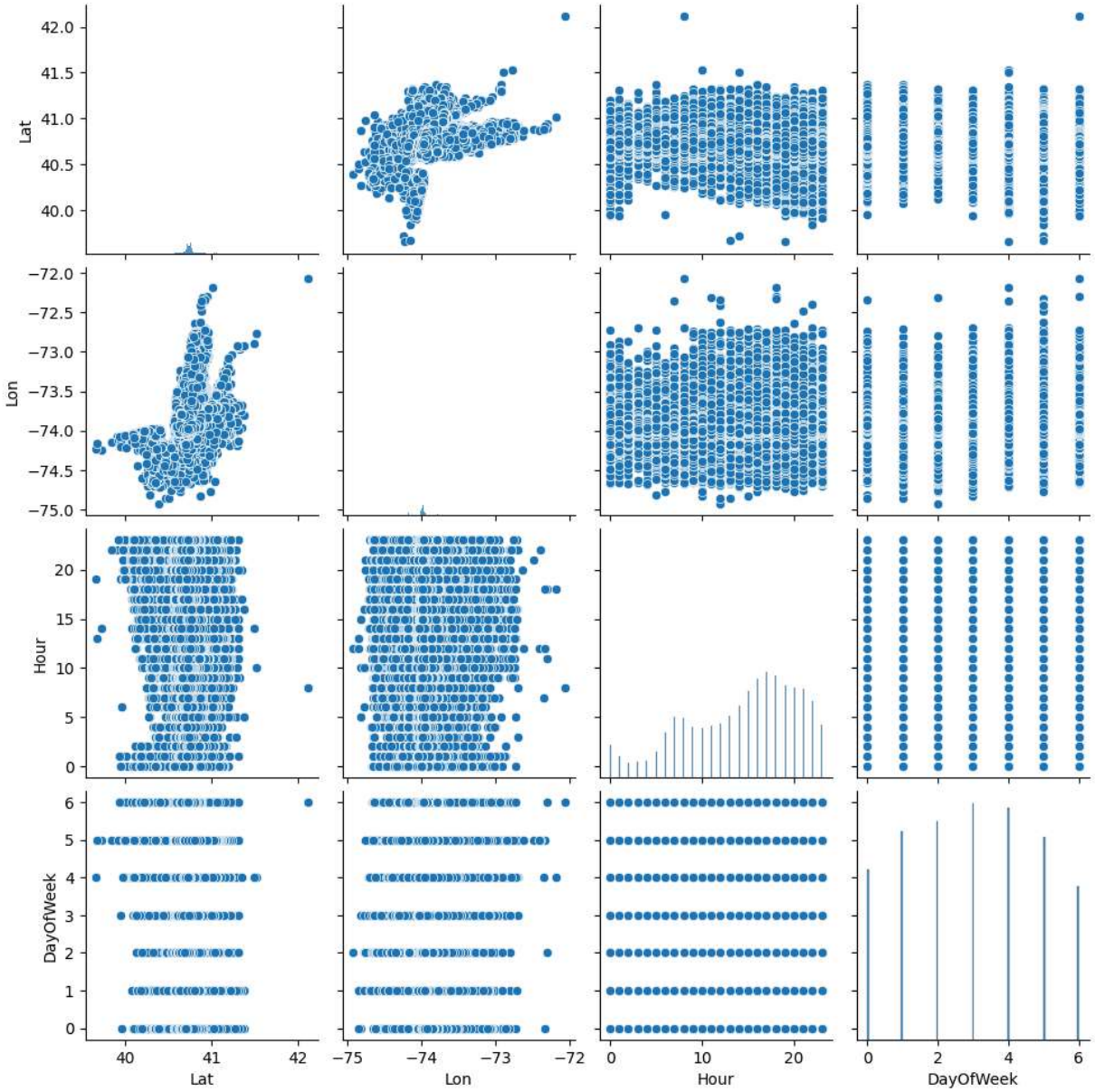
In [29]:
```
print(df["Base Name"])
```

```
0              Unter
1              Unter
2              Unter
3              Unter
4              Unter
             ...
4534322    Danach-NY
4534323    Danach-NY
4534324    Danach-NY
4534325    Danach-NY
4534326    Danach-NY
Name: Base Name, Length: 4534327, dtype: object
```

In [30]:

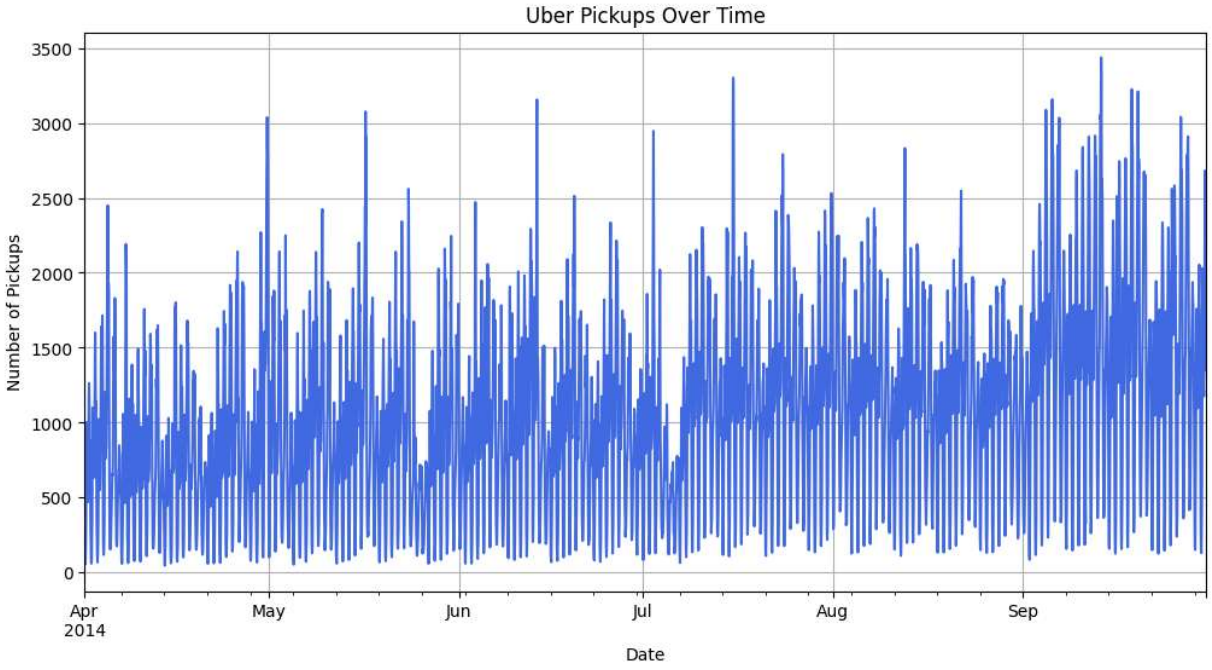
```
sns.pairplot(df[['Lat', 'Lon', 'Hour', 'DayOfWeek']])
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



In [31]:
```
time_series = df.resample('H', on='DateTime').size()

# Plot the time series
plt.figure(figsize=(12, 6))
time_series.plot(title="Uber Pickups Over Time", color="royalblue")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.grid()
plt.show()
```

```
<ipython-input-31-fdbe3ad87e47>:1: FutureWarning: 'H' is deprecated and will be remo
ved in a future version, please use 'h' instead.
  time_series = df.resample('H', on='DateTime').size()
```



In [32]:
```
result = adfuller(time_series)
print(f"ADF Statistic: {result[0]}")
print(f"P-Value: {result[1]}")

if result[1] < 0.05:
    print("The time series is stationary.")
else:
    print("The time series is NOT stationary. Differencing needed.")
```

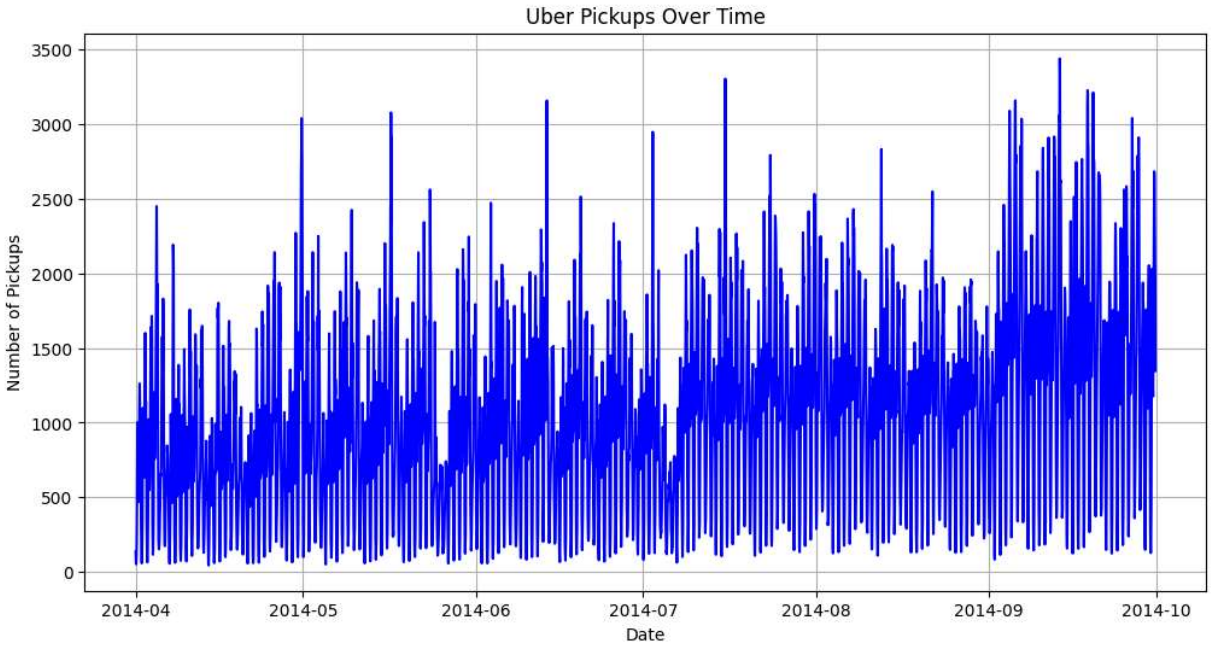
```
ADF Statistic: -4.906252894492546
P-Value: 3.391006185776848e-05
The time series is stationary.
```

5. Time Series Modelling

Prelim steps

In [33]:
```
df_time_series = df.resample("H", on="DateTime").size().reset_index()
df_time_series.columns = ["DateTime", "Pickups"]

# Plot the time series
plt.figure(figsize=(12, 6))
sns.lineplot(x=df_time_series["DateTime"], y=df_time_series["Pickups"], color="blue
plt.title("Uber Pickups Over Time")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.grid()
plt.show()
```

```
<ipython-input-33-caa9d0854bbb>:1: FutureWarning: 'H' is deprecated and will be remo
ved in a future version, please use 'h' instead.
  df_time_series = df.resample("H", on="DateTime").size().reset_index()
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



In [34]:
```
train_size = int(len(df_time_series) * 0.8)
train, test = df_time_series["Pickups"][:train_size], df_time_series["Pickups"][tra
```

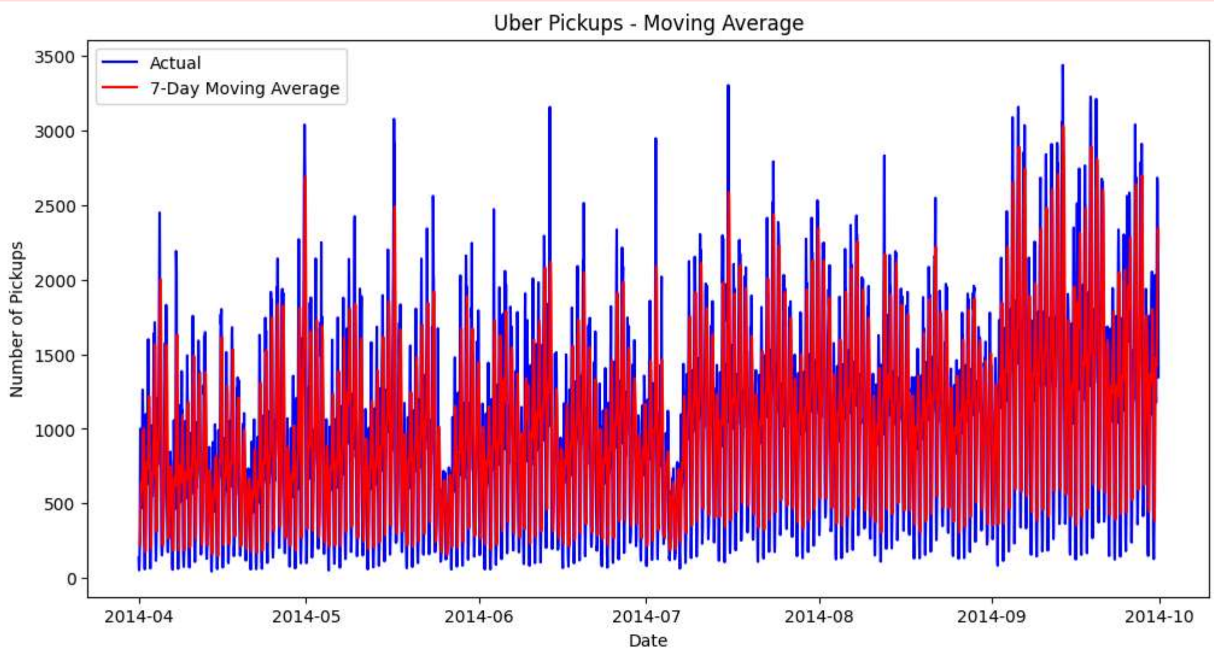
In [35]:
```
def evaluate_forecast(y_true, y_pred, model_name="Model"):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    print(f"{model_name} Evaluation:")
    print(f"MAE: {mae:.2f}")
    print(f"MSE: {mse:.2f}")
    print(f"RMSE: {rmse:.2f}")
    print(f"MAPE: {mape:.2f}%\n")
```

5.1 Moving Average Baseline model

In [36]:
```
df_time_series["MA_7"] = df_time_series["Pickups"].rolling(window=7).mean()

plt.figure(figsize=(12, 6))
sns.lineplot(x=df_time_series["DateTime"], y=df_time_series["Pickups"], label="Actu
sns.lineplot(x=df_time_series["DateTime"], y=df_time_series["MA_7"], label="7-Day M
plt.title("Uber Pickups - Moving Average")
```

```
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.legend()
plt.show()
```

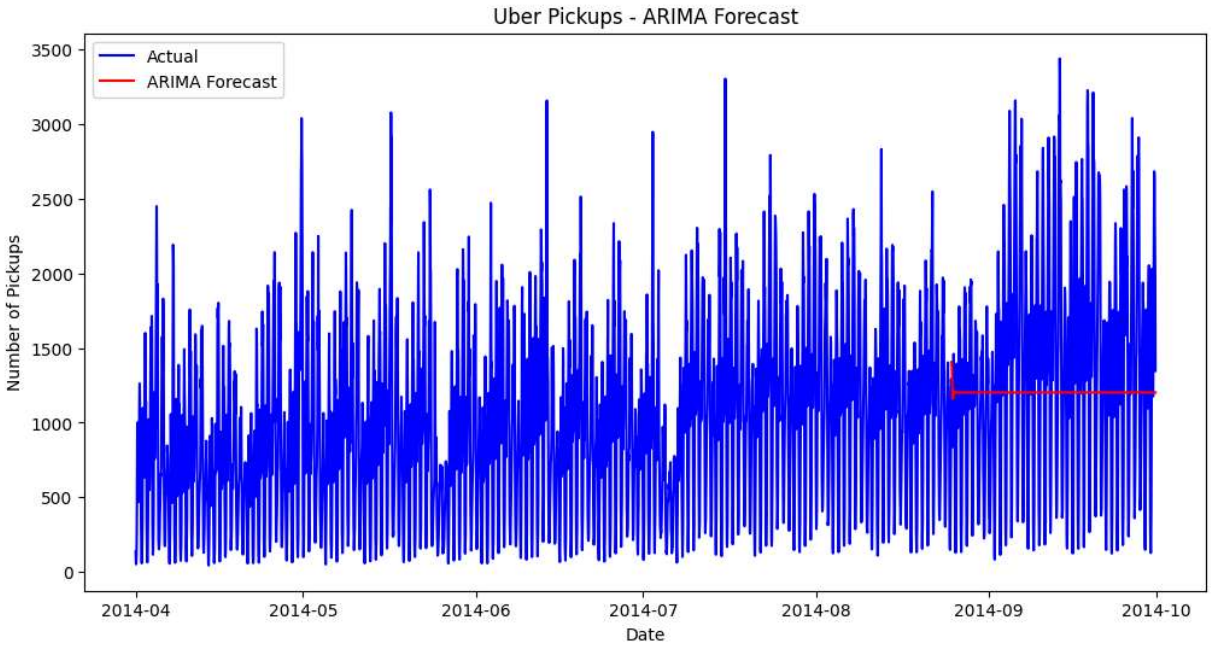
```
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use
_inf_as_na option is deprecated and will be removed in a future version. Convert inf
values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



5.2 Forecasting with ARIMA

In [37]:
```
arima_m = ARIMA(train, order=(5,1,0))
model_fit = arima_m.fit()

forecast = model_fit.forecast(steps=len(test))

forecast_index = df_time_series["DateTime"][train_size:train_size + len(forecast)]

plt.figure(figsize=(12, 6))
plt.plot(df_time_series["DateTime"], df_time_series["Pickups"], label="Actual", col
plt.plot(forecast_index, forecast, label="ARIMA Forecast", color="red")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
```

```
plt.title("Uber Pickups - ARIMA Forecast")
plt.legend()
plt.show()
```

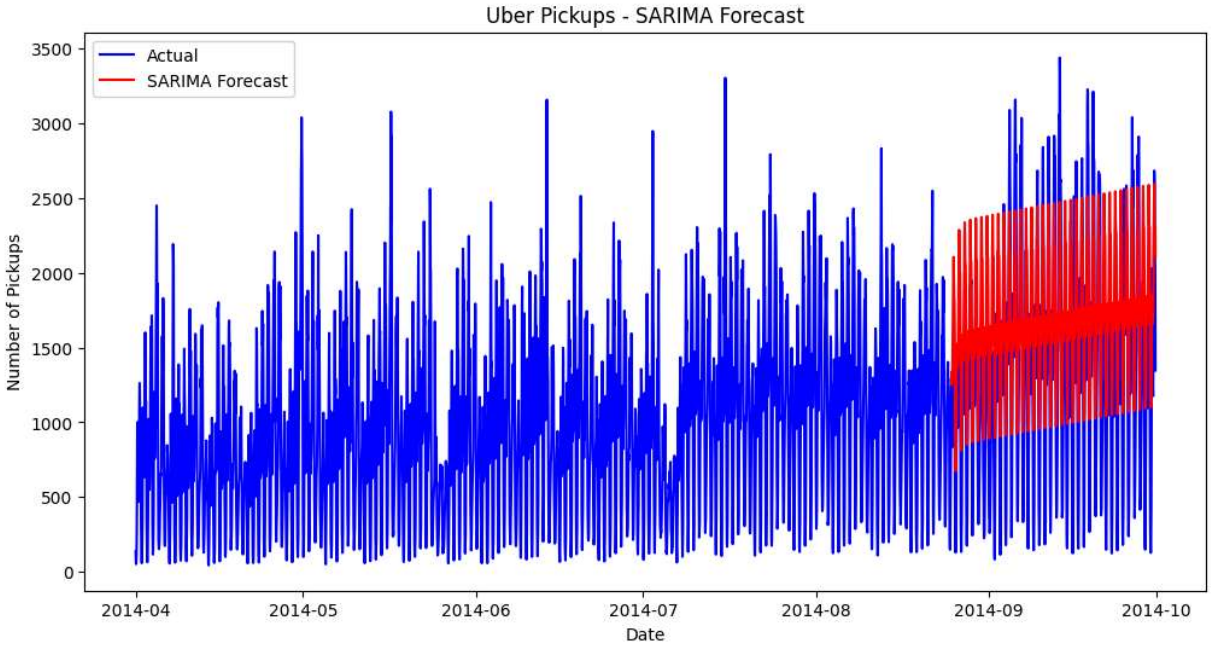


In [38]:
```
evaluate_forecast(test, forecast, "ARIMA")
```

```
ARIMA Evaluation:
MAE: 623.88
MSE: 597843.26
RMSE: 773.20
MAPE: 91.84%
```

5.2 Forecasting with SARIMA

In [39]:
```
sarima_m = SARIMAX(train, order=(1,1,1), seasonal_order=(1,1,1,24))
sarima_fit = sarima_m.fit()

forecast = sarima_fit.get_forecast(steps=len(test)).predicted_mean

forecast_index = df_time_series["DateTime"][train_size:train_size + len(forecast)]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(df_time_series["DateTime"], df_time_series["Pickups"], label="Actual", col
plt.plot(forecast_index, forecast, label="SARIMA Forecast", color="red")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.title("Uber Pickups - SARIMA Forecast")
plt.legend()
plt.show()
```

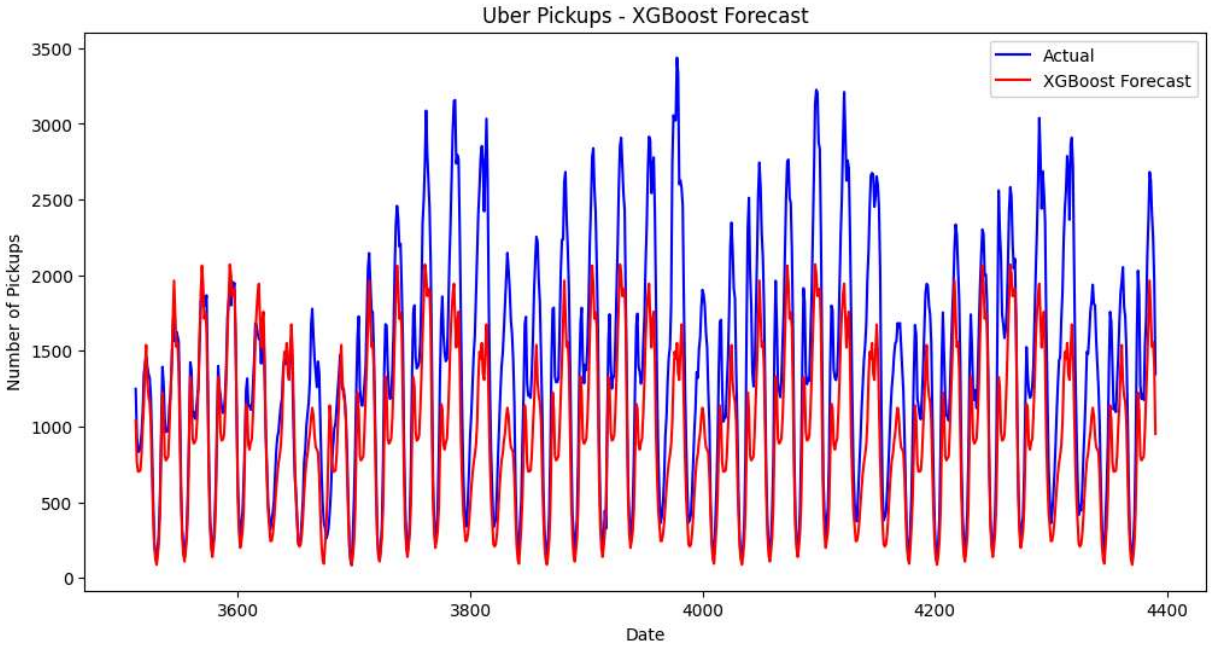


In [40]:
```
evaluate_forecast(test, forecast, "SARIMA")
```

```
SARIMA Evaluation:
MAE: 474.58
MSE: 312986.74
RMSE: 559.45
MAPE: 81.67%
```

5.4 Forecasting with XGBoost

In [41]:
```
df_time_series["Hour"] = df_time_series["DateTime"].dt.hour
df_time_series["DayOfWeek"] = df_time_series["DateTime"].dt.dayofweek

X = df_time_series[["Hour", "DayOfWeek"]]
y = df_time_series["Pickups"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Fa

xgb_model = XGBRegressor(n_estimators=100, learning_rate=0.1)
xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_test)

plt.figure(figsize=(12, 6))
plt.plot(y_test.index, y_test, label="Actual", color="blue")
plt.plot(y_test.index, y_pred, label="XGBoost Forecast", color="red")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.title("Uber Pickups - XGBoost Forecast")
plt.legend()
plt.show()
```



In [42]:
```
evaluate_forecast(y_test, y_pred, "XGBoost")
```

```
XGBoost Evaluation:
MAE: 429.28
MSE: 296126.57
RMSE: 544.18
MAPE: 30.76%
```

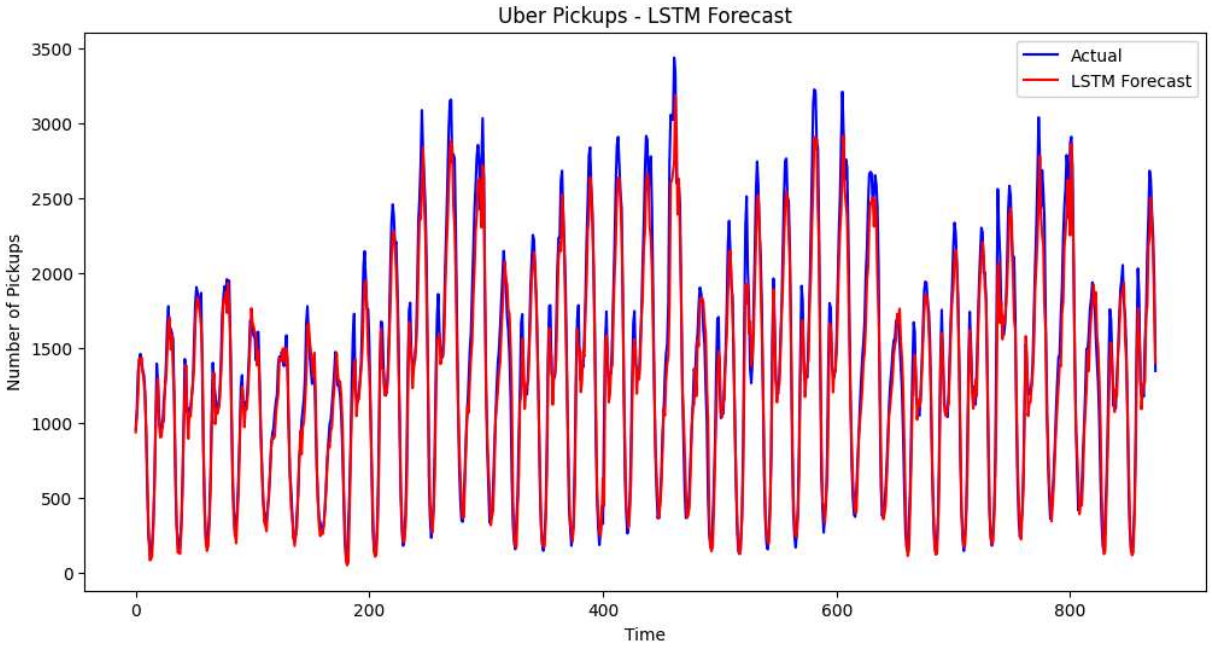
5.3 Forecasting with LSTM

In [43]:
```
scaler = MinMaxScaler()
df_time_series["Scaled_Pickups"] = scaler.fit_transform(df_time_series["Pickups"].v

def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

seq_length = 24
X, y = create_sequences(df_time_series["Scaled_Pickups"].values, seq_length)

split = int(0.8 * len(X))
X_train, X_test, y_train, y_test = X[:split], X[split:], y[:split], y[split:]

model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(seq_length, 1))
    LSTM(50, activation='relu'),
    Dense(1)
])

model.compile(optimizer="adam", loss="mse")
```

```
model.fit(X_train, y_train, epochs=20, batch_size=16, validation_data=(X_test, y_te

y_pred = model.predict(X_test)

y_pred_inv = scaler.inverse_transform(y_pred)
y_test_inv = scaler.inverse_transform(y_test.reshape(-1,1))

plt.figure(figsize=(12, 6))
plt.plot(y_test_inv, label="Actual", color="blue")
plt.plot(y_pred_inv, label="LSTM Forecast", color="red")
plt.xlabel("Time")
plt.ylabel("Number of Pickups")
plt.title("Uber Pickups - LSTM Forecast")
plt.legend()
plt.show()
```

```
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarnin
g: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequenti
al models, prefer using an `Input(shape)` object as the first layer in the model ins
tead.
  super().__init__(**kwargs)
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 19ms/step - loss: 0.0355 - val_loss: 0.0205
Epoch 2/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.0096 - val_loss: 0.0082
Epoch 3/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0066 - val_loss: 0.0044
Epoch 4/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 19ms/step - loss: 0.0041 - val_loss: 0.0032
Epoch 5/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.0030 - val_loss: 0.0066
Epoch 6/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0029 - val_loss: 0.0027
Epoch 7/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0025 - val_loss: 0.0024
Epoch 8/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.0026 - val_loss: 0.0022
Epoch 9/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0024 - val_loss: 0.0025
Epoch 10/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.0023 - val_loss: 0.0022
Epoch 11/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0025 - val_loss: 0.0043
Epoch 12/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0023 - val_loss: 0.0021
Epoch 13/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0019 - val_loss: 0.0020
Epoch 14/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0021 - val_loss: 0.0041
Epoch 15/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0022 - val_loss: 0.0028
Epoch 16/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0021 - val_loss: 0.0024
Epoch 17/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0021 - val_loss: 0.0023
Epoch 18/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0021 - val_loss: 0.0022
Epoch 19/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0019 - val_loss: 0.0022
Epoch 20/20
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0018 - val_loss: 0.0028
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step
```



In [44]:
```
evaluate_forecast(y_test_inv, y_pred_inv, "LSTM")
```

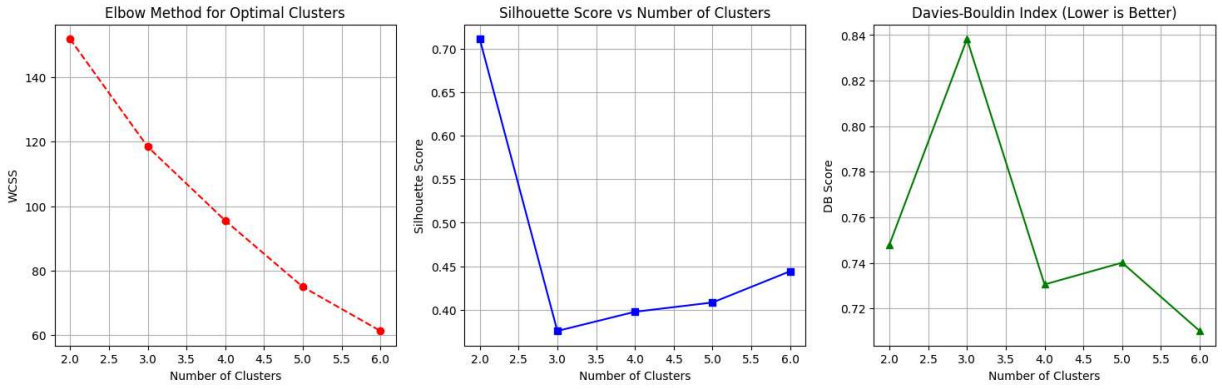
```
LSTM Evaluation:
MAE: 134.53
MSE: 32261.12
RMSE: 179.61
MAPE: 12.19%
```

6. Clustering Modelling for Hotspot Detection

6.1 K Means Clustering

In [45]:
```
df_sampled = df[['Lat', 'Lon']].sample(50000, random_state=42)  # Smaller sample si

# Lists to store evaluation metrics
wcss = []
silhouette_scores = []
db_scores = []
ch_scores = []

for i in range(2, 7):
    kmeans = KMeans(n_clusters=i, random_state=42, n_init=10)
    labels = kmeans.fit_predict(df_sampled)

    # Compute metrics
    wcss.append(kmeans.inertia_)
    silhouette_scores.append(silhouette_score(df_sampled, labels))
    db_scores.append(davies_bouldin_score(df_sampled, labels))
    ch_scores.append(calinski_harabasz_score(df_sampled, labels))

fig, ax = plt.subplots(1, 3, figsize=(18, 5))
```

```
# WCSS (Elbow Method)
ax[0].plot(range(2, 7), wcss, marker='o', linestyle='--', color='red')
ax[0].set_title("Elbow Method for Optimal Clusters")
ax[0].set_xlabel("Number of Clusters")
ax[0].set_ylabel("WCSS")
ax[0].grid()

# Silhouette Score
ax[1].plot(range(2, 7), silhouette_scores, marker='s', linestyle='-', color='blue')
ax[1].set_title("Silhouette Score vs Number of Clusters")
ax[1].set_xlabel("Number of Clusters")
ax[1].set_ylabel("Silhouette Score")
ax[1].grid()

# Davies-Bouldin Score
ax[2].plot(range(2, 7), db_scores, marker='^', linestyle='-', color='green')
ax[2].set_title("Davies-Bouldin Index (Lower is Better)")
ax[2].set_xlabel("Number of Clusters")
ax[2].set_ylabel("DB Score")
ax[2].grid()

plt.show()
```

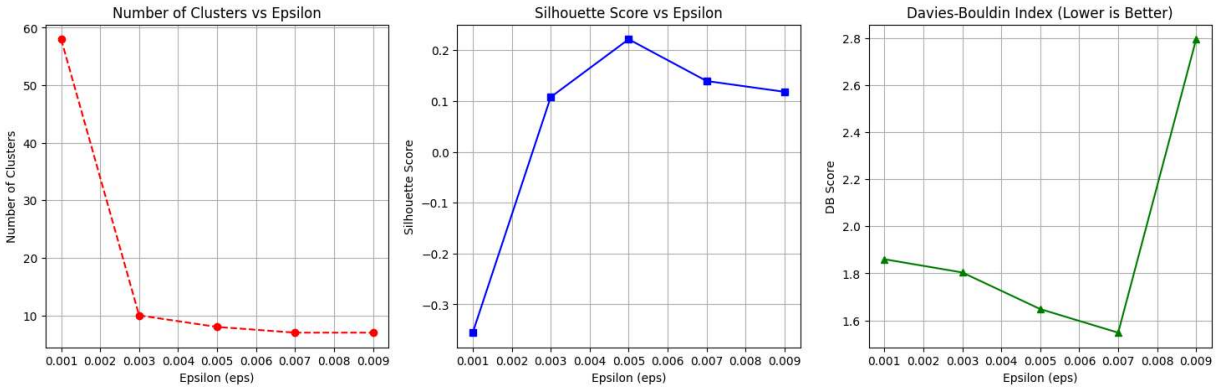


In [46]:
```
best_k = np.argmax(silhouette_scores) + 2

kmeans = KMeans(n_clusters=best_k, random_state=42, n_init=10)
labels = kmeans.fit_predict(df_sampled)

sil_score = silhouette_score(df_sampled, labels)
db_score = davies_bouldin_score(df_sampled, labels)
ch_score = calinski_harabasz_score(df_sampled, labels)

print(f"Optimized Evaluation for Best K ({best_k} Clusters):")
print(f"Silhouette Score: {sil_score:.2f}")
print(f"Davies-Bouldin Index: {db_score:.2f}")
print(f"Calinski-Harabasz Index: {ch_score:.2f}")
```

```
Optimized Evaluation for Best K (2 Clusters):
Silhouette Score: 0.71
Davies-Bouldin Index: 0.75
Calinski-Harabasz Index: 30268.16
```

6.2 Density-Based Clustering (DBSCAN)

In [47]:
```
df_sampled = df[['Lat', 'Lon']].sample(10000, random_state=42)

eps_values = np.arange(0.001, 0.01, 0.002)
silhouette_scores = []
db_scores = []
num_clusters = []

for eps in eps_values:
    dbscan = DBSCAN(eps=eps, min_samples=20)
    labels = dbscan.fit_predict(df_sampled)

    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    num_clusters.append(n_clusters)

    if n_clusters > 1:
        silhouette_scores.append(silhouette_score(df_sampled, labels))
        db_scores.append(davies_bouldin_score(df_sampled, labels))
    else:
        silhouette_scores.append(np.nan)  # Avoids NaN errors
        db_scores.append(np.nan)

# Plot Evaluation Metrics
fig, ax = plt.subplots(1, 3, figsize=(18, 5))

# Number of Clusters vs Epsilon
ax[0].plot(eps_values, num_clusters, marker='o', linestyle='--', color='red')
ax[0].set_title("Number of Clusters vs Epsilon")
ax[0].set_xlabel("Epsilon (eps)")
ax[0].set_ylabel("Number of Clusters")
ax[0].grid()

# Silhouette Score
ax[1].plot(eps_values, silhouette_scores, marker='s', linestyle='-', color='blue')
ax[1].set_title("Silhouette Score vs Epsilon")
ax[1].set_xlabel("Epsilon (eps)")
ax[1].set_ylabel("Silhouette Score")
ax[1].grid()

# Davies-Bouldin Score
ax[2].plot(eps_values, db_scores, marker='^', linestyle='-', color='green')
ax[2].set_title("Davies-Bouldin Index (Lower is Better)")
ax[2].set_xlabel("Epsilon (eps)")
ax[2].set_ylabel("DB Score")
ax[2].grid()

plt.show()
```



In [48]:
```
best_eps_idx = np.nanargmax(silhouette_scores)
best_eps = eps_values[best_eps_idx]

dbscan = DBSCAN(eps=best_eps, min_samples=20)
labels = dbscan.fit_predict(df_sampled)

n_clusters = len(set(labels)) - (1 if -1 in labels else 0)

if n_clusters > 1:
    sil_score = silhouette_score(df_sampled, labels)
    db_score = davies_bouldin_score(df_sampled, labels)
else:
    sil_score = np.nan
    db_score = np.nan

print(f"Optimized Evaluation for DBSCAN (Epsilon = {best_eps:.4f}):")
print(f"Number of Clusters: {n_clusters}")
print(f"Silhouette Score: {sil_score:.2f}")
print(f"Davies-Bouldin Index: {db_score:.2f}")
```

```
Optimized Evaluation for DBSCAN (Epsilon = 0.0050):
Number of Clusters: 8
Silhouette Score: 0.22
Davies-Bouldin Index: 1.65
```

7. Classification Model

In [49]:
```
df.head()
```

Out[49]:

	Date/Time	Lat	Lon	Base	DateTime	Base Name	Year	Month	Day	Hour	Mir
0	5/1/2014 0:02:00	40.7521	-73.9914	B02512	2014-05-01 00:02:00	Unter	2014	5	1	0	
1	5/1/2014 0:06:00	40.6965	-73.9715	B02512	2014-05-01 00:06:00	Unter	2014	5	1	0	
2	5/1/2014 0:15:00	40.7464	-73.9838	B02512	2014-05-01 00:15:00	Unter	2014	5	1	0	
3	5/1/2014 0:17:00	40.7463	-74.0011	B02512	2014-05-01 00:17:00	Unter	2014	5	1	0	
4	5/1/2014 0:17:00	40.7594	-73.9734	B02512	2014-05-01 00:17:00	Unter	2014	5	1	0	

In [50]:
```
min_time = df["DateTime"].min()
max_time = df["DateTime"].max()

min_lat, max_lat = df["Lat"].min(), df["Lat"].max()
min_lon, max_lon = df["Lon"].min(), df["Lon"].max()
```

In [51]:
```
num_samples = len(df)
```

In [52]:
```
random_lats = np.random.uniform(min_lat, max_lat, num_samples)
random_lons = np.random.uniform(min_lon, max_lon, num_samples)
```

In [53]:
```
random_times = [min_time + timedelta(seconds=np.random.randint(0, int((max_time - m
                for _ in range(num_samples)]
```

In [54]:
```
negative_samples = pd.DataFrame({
    "Lat": random_lats,
    "Lon": random_lons,
    "DateTime": random_times,
    "pickup": 0
})
```

In [55]:
```
df["pickup"] = 1
```

In [56]:
```
positive_df = df[['Lat','Lon','DateTime','pickup']]
```

In [57]:
```
final_df = pd.concat([positive_df, negative_samples], ignore_index=True)
```

In [58]:
```
final_df = final_df.sample(frac=1).reset_index(drop=True)
```

In [59]:
```
final_df.head()
```

Out[59]:

	Lat	Lon	DateTime	pickup
0	41.955156	-74.352548	2014-09-11 15:40:09	0
1	40.766400	-73.967100	2014-05-20 15:22:00	1
2	41.678414	-73.783546	2014-05-24 12:33:50	0
3	40.714900	-73.996700	2014-09-07 14:48:00	1
4	40.742500	-74.000500	2014-05-10 17:10:00	1

In [60]:
```
X = final_df[['Lat','Lon']]
Y=final_df[['pickup']]
```

In [61]:
```
train_x, test_x, train_y, test_y = train_test_split(X,Y, test_size=0.2,random_state
```

In [62]:
```
def evaluate_classification(y_true, y_pred, model_name="Model"):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    auc_roc = roc_auc_score(y_true, y_pred)

    print(f"{model_name} Evaluation:")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 Score: {f1:.2f}")
    print(f"AUC-ROC: {auc_roc:.2f}\n")
```

7.1 Logistic Regression

In [63]:
```
lr_model = LogisticRegression()
lr_model.fit(train_x, train_y)
y_pred = lr_model.predict(test_x)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:1143: DataConver
sionWarning: A column-vector y was passed when a 1d array was expected. Please chang
e the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

In [64]:
```
evaluate_classification(test_y,y_pred,"Logistic Regression")
```

```
Logistic Regression Evaluation:
Accuracy: 0.80
Precision: 0.72
Recall: 0.99
F1 Score: 0.83
AUC-ROC: 0.80
```

7.2 Random Forest

In [65]:
```
rf_model = RandomForestClassifier(n_estimators=500)
rf_model.fit(train_x, train_y)
y_pred = rf_model.predict(test_x)
```

```
<ipython-input-65-68ddba18fd22>:2: DataConversionWarning: A column-vector y was pass
ed when a 1d array was expected. Please change the shape of y to (n_samples,), for e
xample using ravel().
  rf_model.fit(train_x, train_y)
```

In [66]:
```
evaluate_classification(test_y,y_pred,"Random Forest")
```

```
Random Forest Evaluation:
Accuracy: 0.99
Precision: 0.99
Recall: 0.99
F1 Score: 0.99
AUC-ROC: 0.99
```

8. Conclusion

For the Time series segment, we shall use the LSTM model which has the lowest MAPE For clustering segment, we shall use the K means model which has the highest Silhouette score among the two The Random Forest model can be used to predict the probability of Uber pickup based on the previous lat-long data

In [67]:
```
model.save("lstm_uber_model.h5")
print("LSTM model saved as 'lstm_uber_model.h5'")
```

```
LSTM model saved as 'lstm_uber_model.h5'
```

In [68]:
```
joblib.dump(kmeans, 'kmeans_uber_model.pkl')
print("K-Means model saved as 'kmeans_uber_model.pkl'")
```

```
K-Means model saved as 'kmeans_uber_model.pkl'
```

In [69]:
```
joblib.dump(kmeans, 'rf_uber_model.pkl')
print("RF model saved as 'rf_uber_model.pkl'")
```

```
RF model saved as 'rf_uber_model.pkl'
```

New York City Uber Time Series Analysis and Ride Demand Prediction

Authors:

Koelgeet Kaur

Based on the dataset (source: https://github.com/fivethirtyeight/uber-tlc-foil-response) we are working on the following objective for this notebook:

Ride Demand Prediction

--- **Goal**: Predict the number of Uber pickups at a given time and location.

--- **Input Features:** Date & Time (hour, day, month, weekday/weekend) Location (Latitude & Longitude or Zone ID)

--- **Output:** Expected number of Uber pickups.

--- **Models:**

1. **Time Series Models:** ARIMA, SARIMA
2. **Machine Learning Models:** XGBoost, Random Forest, LSTM (Deep Learning)

Tableau Public Dashboard Links

https://public.tableau.com/app/profile/koelgeet.kaur/viz/UberDataAnalysis_17402429676560/Da

https://public.tableau.com/app/profile/koelgeet.kaur/viz/UberDataAnalysis_17402429676560/Da

Introduction

This analysis utilizes data on over 4.5 million Uber pickups in New York City from April to September 2014. The data is organized into six files, separated by month, and includes the following columns:

Date/Time: The date and time of the Uber pickup

Lat: The latitude of the Uber pickup

Lon: The longitude of the Uber pickup

Base: The TLC base company code affiliated with the Uber pickup

As we do not have access to revenue or fee per trip information, our analysis will focus on analyzing user patterns in date and time to gain insights.

1. Importing Libraries

In [ ]:
```
import pandas as pd
import numpy as np
import glob
import geopandas as gpd
import matplotlib.pyplot as plt
import seaborn as sns
import folium
from folium.plugins import HeatMap
import plotly.express as px
from shapely.geometry import Point
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from prophet import Prophet
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
import joblib
from tensorflow.keras.models import load_model
import tensorflow.keras.losses
import datetime
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
```

2. Importing dataset

In [ ]:
```
# Kaggle dataset paths
data_apr = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-apr14.csv')
data_may = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-may14.csv')
data_jun = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-jun14.csv')
data_jul = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-jul14.csv')
data_aug = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-aug14.csv')
data_sep = pd.read_csv('/kaggle/input/uber-nyc-2014/uber-raw-data-sep14.csv')

uber_data = pd.concat([data_apr,data_may,data_jun, data_jul,data_aug,data_sep])
```

3. Data Pre-processing

In [ ]:
```
uber_data.head()
```

Out[ ]:

	Date/Time	Lat	Lon	Base
0	4/1/2014 0:11:00	40.7690	-73.9549	B02512
1	4/1/2014 0:17:00	40.7267	-74.0345	B02512
2	4/1/2014 0:21:00	40.7316	-73.9873	B02512
3	4/1/2014 0:28:00	40.7588	-73.9776	B02512
4	4/1/2014 0:33:00	40.7594	-73.9722	B02512

In [ ]:
```
# Convert Date/Time column to datetime
uber_data.rename(columns={'Date/Time': 'datetime'}, inplace=True)
uber_data['datetime'] = pd.to_datetime(uber_data['datetime'])

uber_data.head()
```

Out[ ]:

	datetime	Lat	Lon	Base
0	2014-04-01 00:11:00	40.7690	-73.9549	B02512
1	2014-04-01 00:17:00	40.7267	-74.0345	B02512
2	2014-04-01 00:21:00	40.7316	-73.9873	B02512
3	2014-04-01 00:28:00	40.7588	-73.9776	B02512
4	2014-04-01 00:33:00	40.7594	-73.9722	B02512

In [ ]:
```
uber_data.isna().sum().sum()
```

Out[ ]:
```
0
```

4. Exploratory Data Analysis

4.1 Analyzing the data for each month from April to September 2014

4.1.0 Preliminary steps

In [ ]:
```
##------Renaming the Date/Time column to datetime and changing its data type to dat

data_apr.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_apr['datetime'] = pd.to_datetime(data_apr['datetime'])

data_may.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_may['datetime'] = pd.to_datetime(data_may['datetime'])

data_jun.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_jun['datetime'] = pd.to_datetime(data_jun['datetime'])
```

```
data_jul.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_jul['datetime'] = pd.to_datetime(data_jul['datetime'])

data_aug.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_aug['datetime'] = pd.to_datetime(data_aug['datetime'])

data_sep.rename(columns={'Date/Time': 'datetime'}, inplace=True)
data_sep['datetime'] = pd.to_datetime(data_sep['datetime'])
```

In [ ]:
```
##----Adding three new columns of Date, Hour and day of the week-----##

# For the month of April
data_apr['Date'] = data_apr['datetime'].dt.date
data_apr['Hour'] = data_apr['datetime'].dt.hour
data_apr['Day_of_Week'] = data_apr['datetime'].dt.day_name()

# For the month of May
data_may['Date'] = data_may['datetime'].dt.date
data_may['Hour'] = data_may['datetime'].dt.hour
data_may['Day_of_Week'] = data_may['datetime'].dt.day_name()

# For the month of June
data_jun['Date'] = data_jun['datetime'].dt.date
data_jun['Hour'] = data_jun['datetime'].dt.hour
data_jun['Day_of_Week'] = data_jun['datetime'].dt.day_name()

# For the month of July
data_jul['Date'] = data_jul['datetime'].dt.date
data_jul['Hour'] = data_jul['datetime'].dt.hour
data_jul['Day_of_Week'] = data_jul['datetime'].dt.day_name()

# For the month of August
data_aug['Date'] = data_aug['datetime'].dt.date
data_aug['Hour'] = data_aug['datetime'].dt.hour
data_aug['Day_of_Week'] = data_aug['datetime'].dt.day_name()

# For the month of September
data_sep['Date'] = data_sep['datetime'].dt.date
data_sep['Hour'] = data_sep['datetime'].dt.hour
data_sep['Day_of_Week'] = data_sep['datetime'].dt.day_name()
```

In [ ]:
```
##----Calculating the ride demands based on Dates, Hours and Days of the week-----#

# For the month of April
date_counts_apr = data_apr['Date'].value_counts().sort_index()
hour_counts_apr = data_apr['Hour'].value_counts().sort_index()
day_counts_apr = data_apr['Day_of_Week'].value_counts()

# For the month of May
date_counts_may = data_may['Date'].value_counts().sort_index()
hour_counts_may = data_may['Hour'].value_counts().sort_index()
day_counts_may = data_may['Day_of_Week'].value_counts()

# For the month of June
```

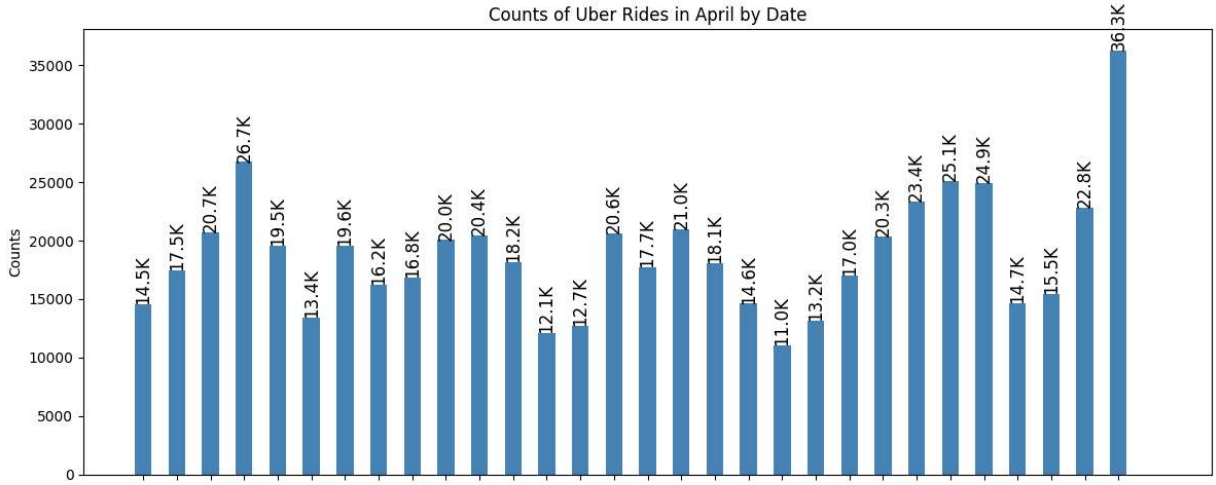
```
date_counts_jun = data_jun['Date'].value_counts().sort_index()
hour_counts_jun = data_jun['Hour'].value_counts().sort_index()
day_counts_jun = data_jun['Day_of_Week'].value_counts()

# For the month of July
date_counts_jul = data_jul['Date'].value_counts().sort_index()
hour_counts_jul = data_jul['Hour'].value_counts().sort_index()
day_counts_jul = data_jul['Day_of_Week'].value_counts()

# For the month of August
date_counts_aug = data_aug['Date'].value_counts().sort_index()
hour_counts_aug = data_aug['Hour'].value_counts().sort_index()
day_counts_aug = data_aug['Day_of_Week'].value_counts()

# For the month of September
date_counts_sep = data_sep['Date'].value_counts().sort_index()
hour_counts_sep = data_sep['Hour'].value_counts().sort_index()
day_counts_sep = data_sep['Day_of_Week'].value_counts()
```

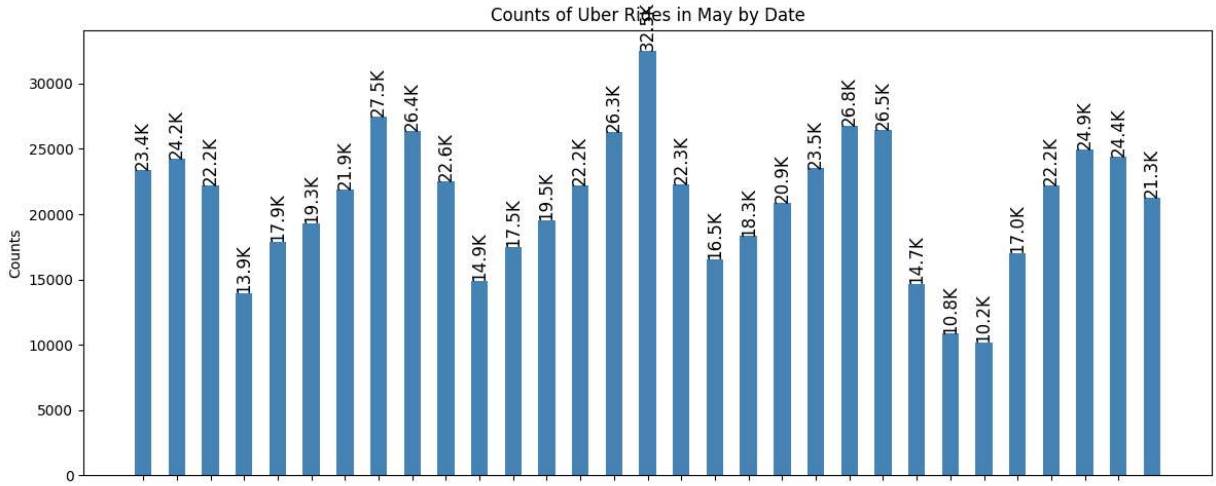
4.1.1 Distribution of Ride Demands by Dates

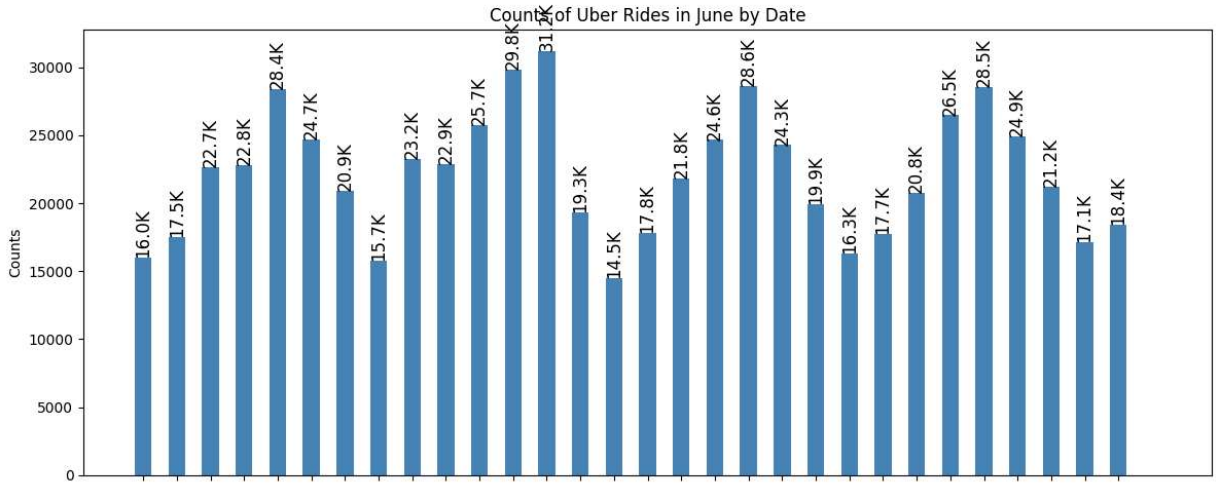
In [ ]:

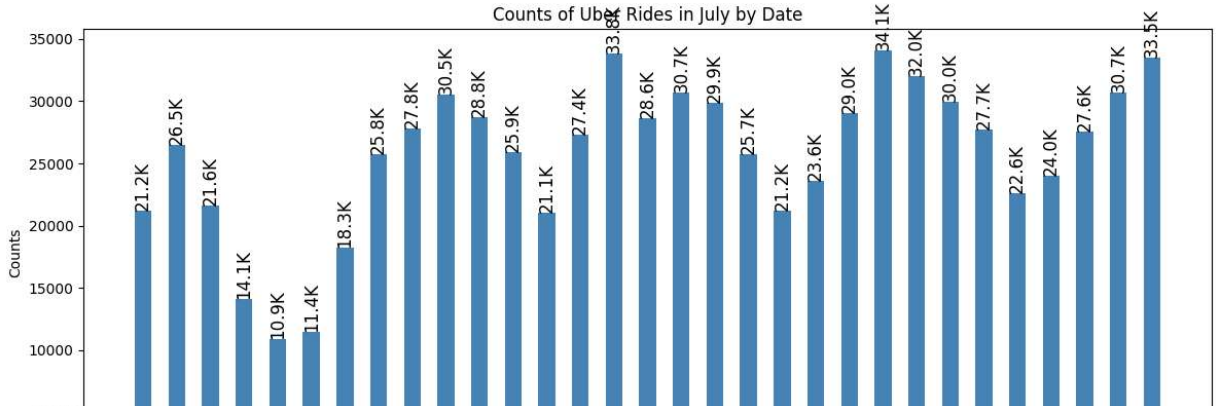
```
# Creating a subplot with 6 rows and 1 column
fig, axes = plt.subplots(nrows=6, ncols=1,figsize=(12,30), sharex=True)

# Months from April to September and their corresponding data
months = ["April", "May", "June", "July", "August", "September"]
date_counts = {
    "April": date_counts_apr,
    "May": date_counts_may,
    "June": date_counts_jun,
    "July": date_counts_jul,
    "August": date_counts_aug,
    "September": date_counts_sep
}

for i, month in enumerate(months):
    ax = axes[i]
    data = date_counts[month]

    # Convert dates to categorical x positions
    x_positions = np.arange(len(data))  # Evenly spaced categorical indices

    # Plot bar chart with reduced width and uniform spacing
    ax.bar(x_positions, data.values, color='steelblue', width=0.5)
    ax.set_ylabel('Counts')
    ax.set_title(f'Counts of Uber Rides in {month} by Date')

    # Reduce the number of x-axis ticks (show every 3rd or 4th date)
    step = 1  # Adjust step size for better spacing
    tick_positions = np.arange(0, len(data), step=step)
    tick_labels = [data.index[j].strftime('%d') for j in tick_positions]

    ax.set_xticks(tick_positions)
    ax.set_xticklabels(tick_labels, rotation=45, fontsize=12)
```

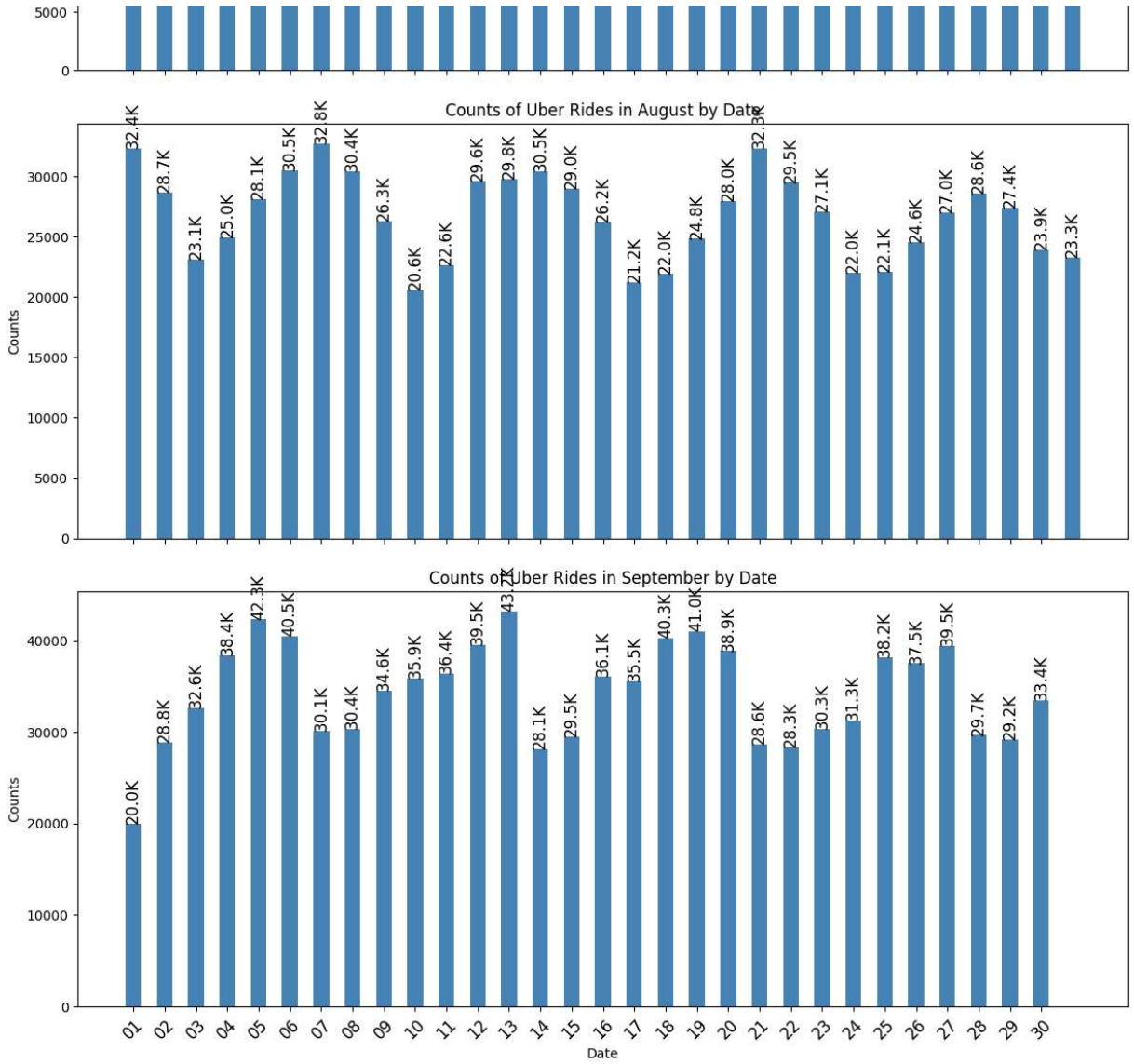
```
    # Add annotations on y-axis
    for j in range(len(data)):
        ax.text(x_positions[j], data.values[j], f"{data.values[j]/1000:.1f}K",
                rotation=90, ha='center', va='bottom', fontsize=12)

# Shared x-label
axes[-1].set_xlabel('Date')

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


Counts of Uber Rides in April by Date
Counts
14.5K
17.5K
20.7K
26.7K
19.5K
13.4K
19.6K
16.2K
16.8K
20.0K
20.4K
18.2K
12.1K
12.7K
20.6K
17.7K
21.0K
18.1K
14.6K
11.0K
13.2K
17.0K
20.3K
23.4K
25.1K
24.9K
14.7K
15.5K
22.8K
36.3K


Counts of Uber Rides in May by Date
Counts
23.4K
24.2K
22.2K
13.9K
17.9K
19.3K
21.9K
27.5K
26.4K
22.6K
14.9K
17.5K
19.5K
22.2K
26.3K
32.5K
22.3K
16.5K
18.3K
20.9K
23.5K
26.8K
26.5K
14.7K
10.8K
10.2K
17.0K
22.2K
24.9K
24.4K
21.3K


Counts of Uber Rides in June by Date
Counts
16.0K
17.5K
22.7K
22.8K
28.4K
24.7K
20.9K
15.7K
23.2K
22.9K
25.7K
29.8K
31.2K
19.3K
14.5K
17.8K
21.8K
24.6K
28.6K
24.3K
19.9K
16.3K
17.7K
20.8K
26.5K
28.5K
24.9K
21.2K
17.1K
18.4K

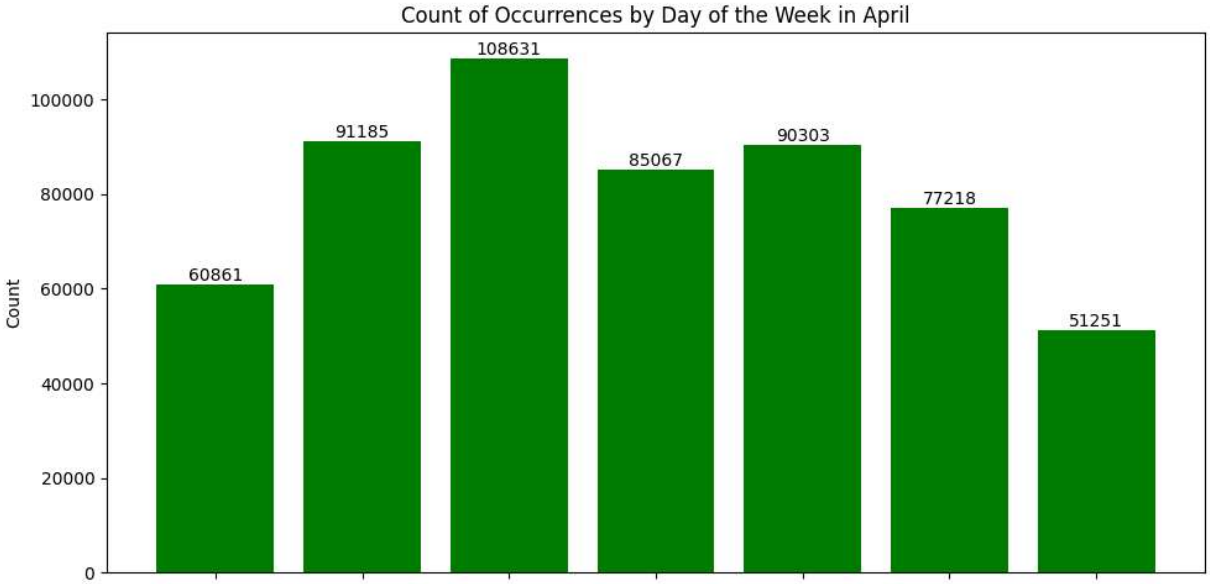

Counts of Uber Rides in July by Date
Counts
21.2K
26.5K
21.6K
14.1K
10.9K
11.4K
18.3K
25.8K
27.8K
30.5K
28.8K
25.9K
21.1K
27.4K
33.8K
28.6K
30.7K
29.9K
25.7K
21.2K
23.6K
29.0K
34.1K
32.0K
30.0K
27.7K
22.6K
24.0K
27.6K
30.7K
33.5K

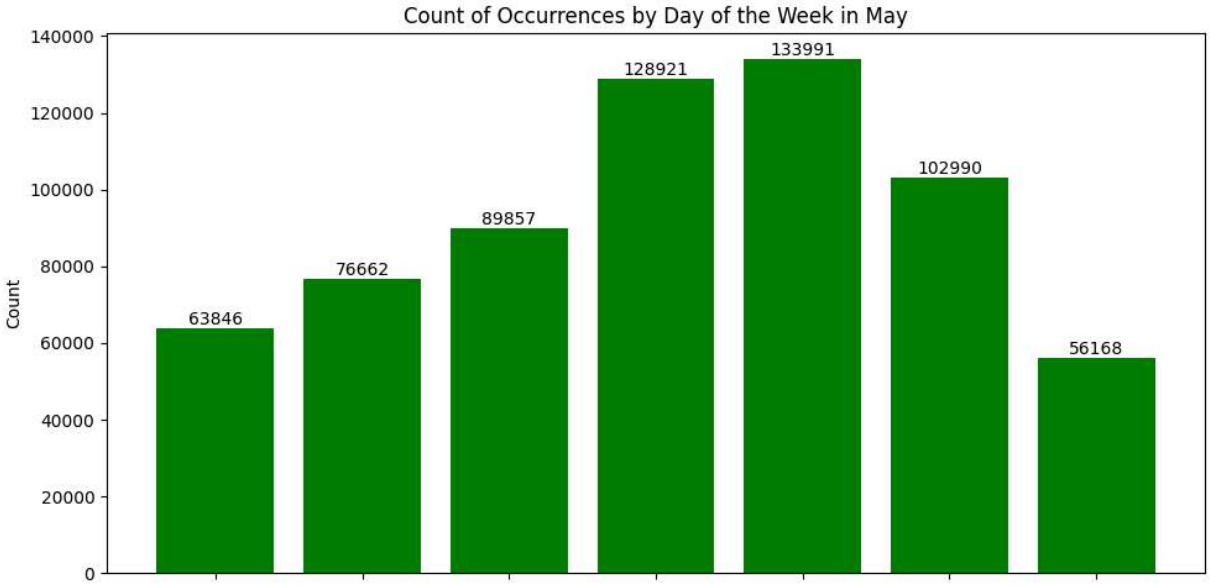


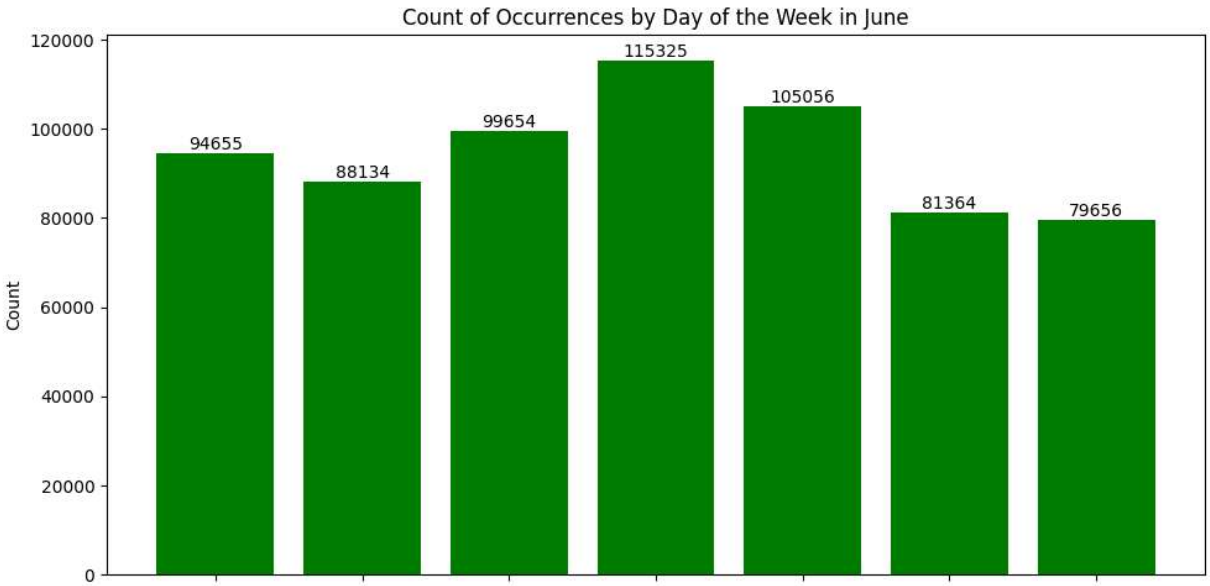
4.1.2 Distribution of Ride Demands by Days of the Week

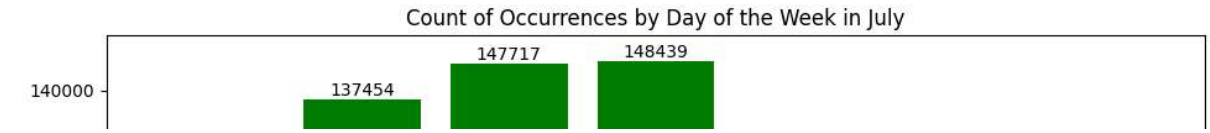
In [ ]:
```
# Define the correct order of days
days_label = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', '

# Reorder day_counts for each month and fill missing values with 0
day_counts_apr = data_apr['Day_of_Week'].value_counts().reindex(days_label)
day_counts_may = data_may['Day_of_Week'].value_counts().reindex(days_label)
day_counts_jun = data_jun['Day_of_Week'].value_counts().reindex(days_label)
day_counts_jul = data_jul['Day_of_Week'].value_counts().reindex(days_label)
day_counts_aug = data_aug['Day_of_Week'].value_counts().reindex(days_label)
day_counts_sep = data_sep['Day_of_Week'].value_counts().reindex(days_label)

fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(10, 30), sharex=True)

# Dictionary with data for each month
months = ["April", "May", "June", "July", "August", "September"]
day_counts_dict = {
    "April": day_counts_apr,
    "May": day_counts_may,
    "June": day_counts_jun,
```

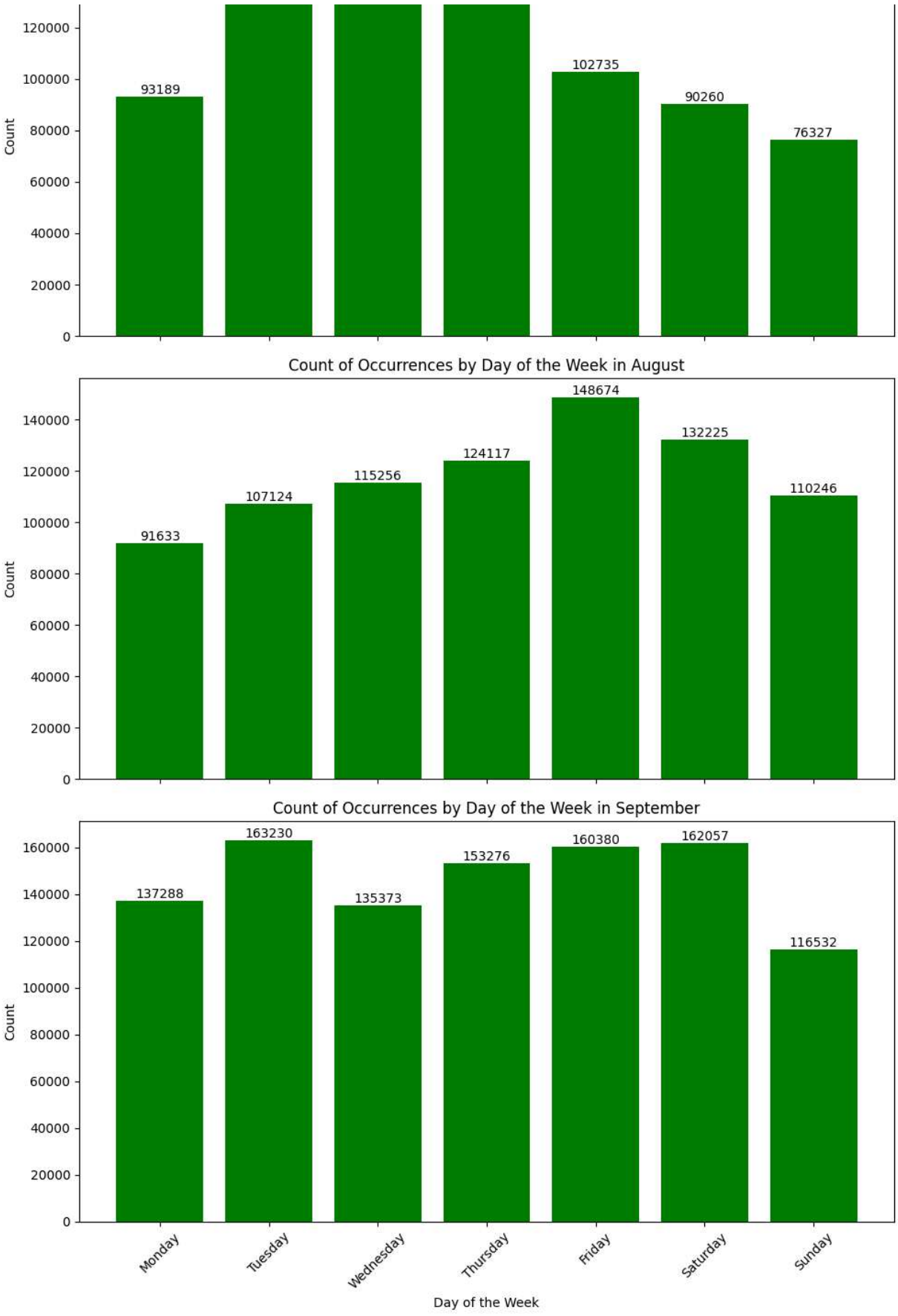
```
    "July": day_counts_jul,
    "August": day_counts_aug,
    "September": day_counts_sep
}

for i, month in enumerate(months):
    ax = axes[i]
    day_counts = day_counts_dict[month]  # Get data for the month

    ax.bar(day_counts.index, day_counts.values, color='green')
    ax.set_ylabel('Count')
    ax.set_title(f'Count of Occurrences by Day of the Week in {month}')

    # Reset x-axis labels
    ax.set_xticks(range(len(days_label)))
    ax.set_xticklabels(days_label, rotation=45)

    # Add annotations on y-axis
    for j, value in enumerate(day_counts.values):
        ax.text(j, value, str(value), ha='center', va='bottom')

# Shared x-label
axes[-1].set_xlabel('Day of the Week')

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


Count of Occurrences by Day of the Week in April
Count
0
20000
40000
60000
80000
100000
60861
91185
108631
85067
90303
77218
51251


Count of Occurrences by Day of the Week in May
Count
0
20000
40000
60000
80000
100000
120000
140000
63846
76662
89857
128921
133991
102990
56168

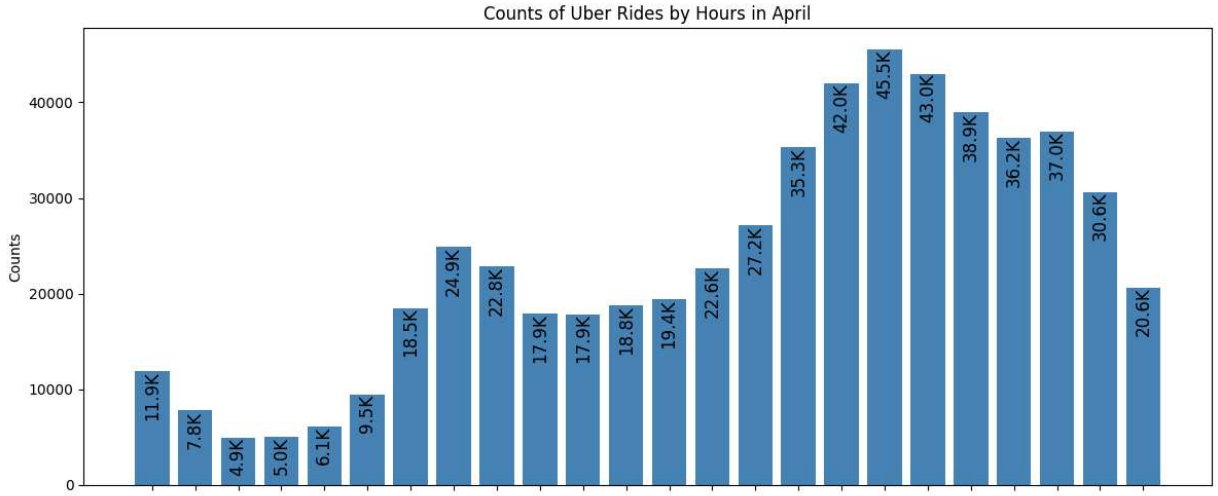

Count of Occurrences by Day of the Week in June
Count
0
20000
40000
60000
80000
100000
120000
94655
88134
99654
115325
105056
81364
79656

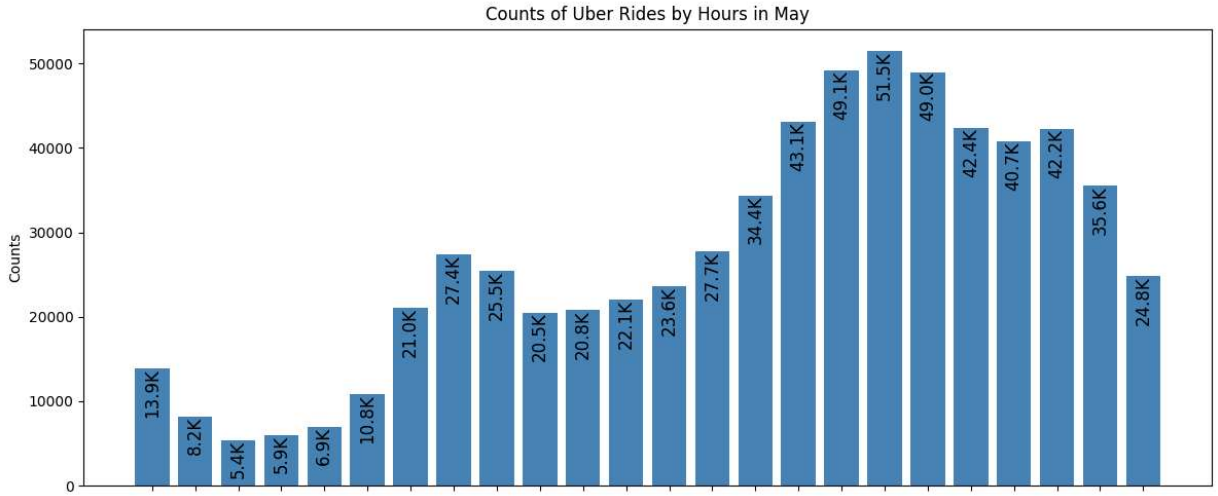

Count of Occurrences by Day of the Week in July
140000
137454
147717
148439

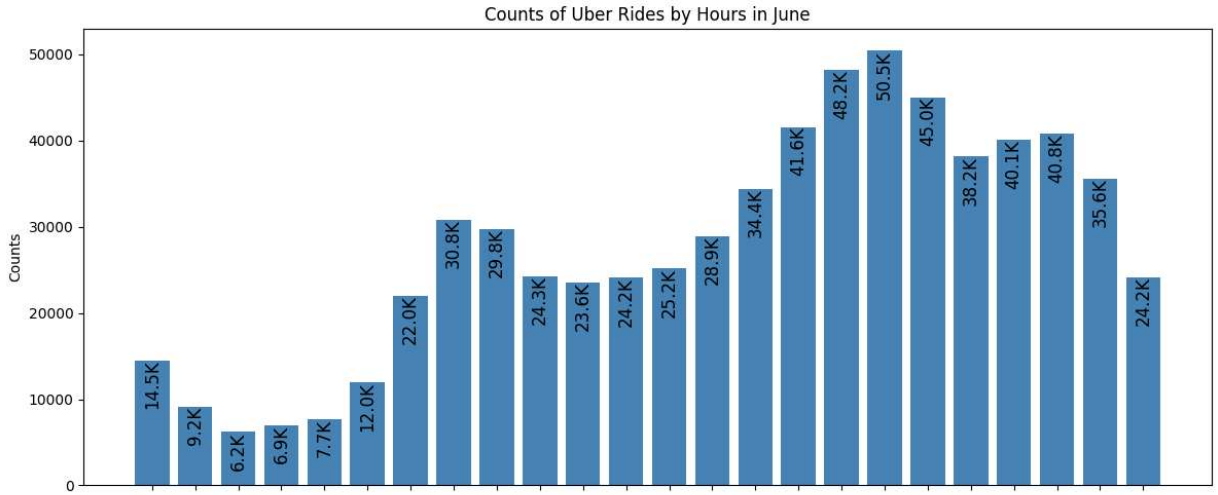


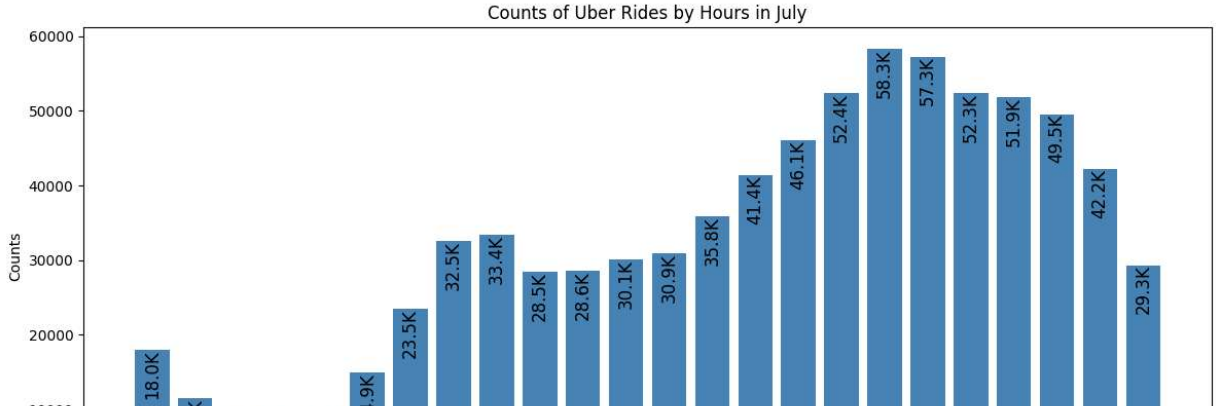
4.1.3 Distribution of Ride Demands by Hours

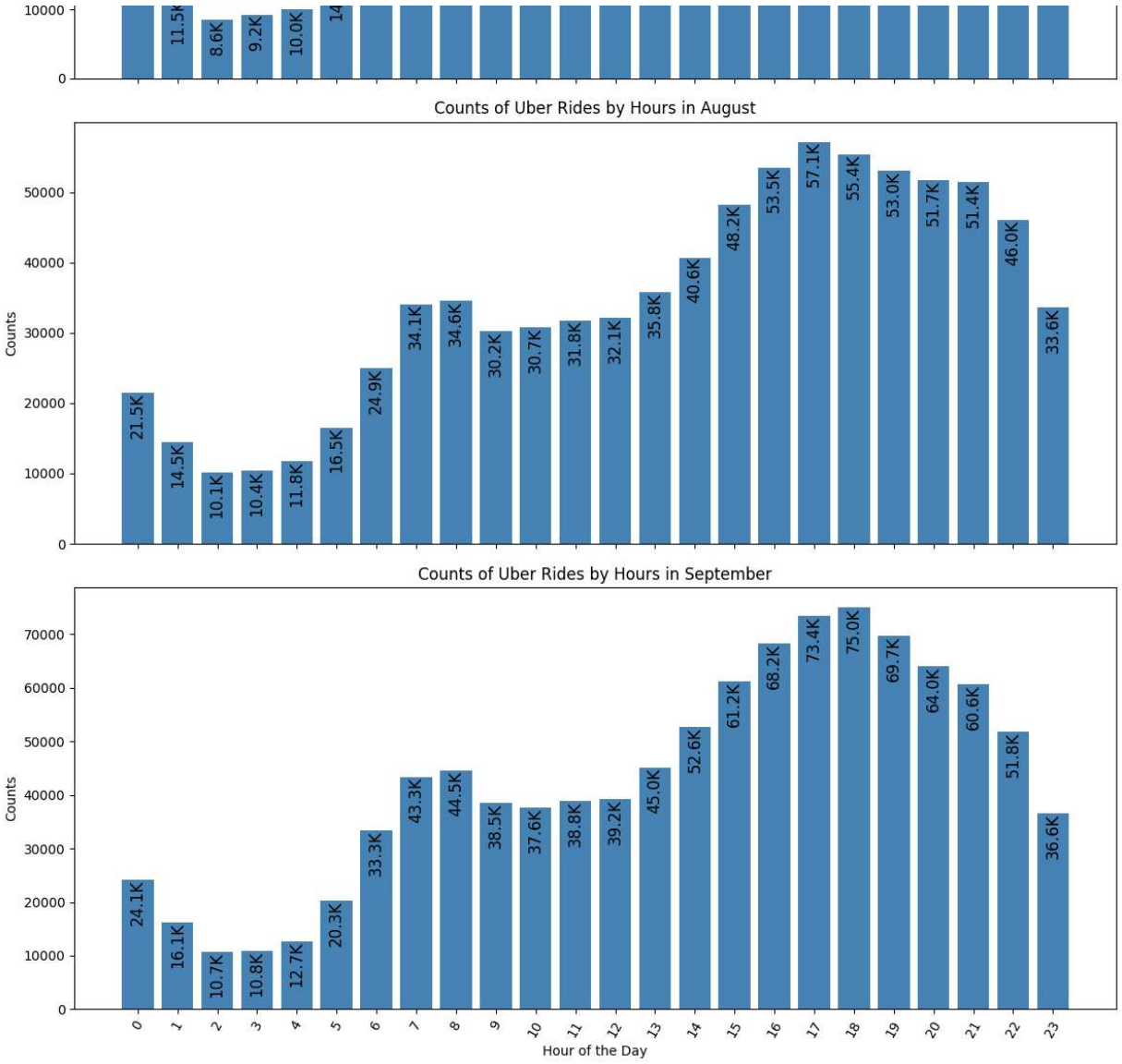
In [ ]:
```
# Creating a subplot with 6 rows and 1 column
fig, axes = plt.subplots(nrows=6, ncols=1, figsize=(12, 30), sharex=True)

# Dictionary with data for each month
months = ["April", "May", "June", "July", "August", "September"]
hour_counts_dict = {
    "April": data_apr['Hour'].value_counts().sort_index(),
    "May": data_may['Hour'].value_counts().sort_index(),
    "June": data_jun['Hour'].value_counts().sort_index(),
    "July": data_jul['Hour'].value_counts().sort_index(),
    "August": data_aug['Hour'].value_counts().sort_index(),
    "September": data_sep['Hour'].value_counts().sort_index()
}

for i, month in enumerate(months):
    ax = axes[i]
    hour_counts = hour_counts_dict[month]

    # Plot bar chart
    ax.bar(hour_counts.index, hour_counts.values, color='steelblue')
    ax.set_ylabel('Counts')
    ax.set_title(f'Counts of Uber Rides by Hours in {month}')

    # Set x-ticks for hours (0-23)
    ax.set_xticks(hour_counts.index)
    ax.set_xticklabels(hour_counts.index, rotation=60)

    # Add annotations on y-axis
    for j in range(len(hour_counts)):
        ax.text(hour_counts.index[j], hour_counts.values[j], f"{hour_counts.values[
                rotation=90, ha='center', va='top', fontsize=12)

# Shared x-label
axes[-1].set_xlabel('Hour of the Day')

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


Counts of Uber Rides by Hours in April
Counts
0
10000
20000
30000
40000
11.9K
7.8K
4.9K
5.0K
6.1K
9.5K
18.5K
24.9K
22.8K
17.9K
17.9K
18.8K
19.4K
22.6K
27.2K
35.3K
42.0K
45.5K
43.0K
38.9K
36.2K
37.0K
30.6K
20.6K


Counts of Uber Rides by Hours in May
Counts
0
10000
20000
30000
40000
50000
13.9K
8.2K
5.4K
5.9K
6.9K
10.8K
21.0K
27.4K
25.5K
20.5K
20.8K
22.1K
23.6K
27.7K
34.4K
43.1K
49.1K
51.5K
49.0K
42.4K
40.7K
42.2K
35.6K
24.8K


Counts of Uber Rides by Hours in June
Counts
0
10000
20000
30000
40000
50000
14.5K
9.2K
6.2K
6.9K
7.7K
12.0K
22.0K
30.8K
29.8K
24.3K
23.6K
24.2K
25.2K
28.9K
34.4K
41.6K
48.2K
50.5K
45.0K
38.2K
40.1K
40.8K
35.6K
24.2K


Counts of Uber Rides by Hours in July
Counts
0
10000
20000
30000
40000
50000
60000
18.0K
23.5K
32.5K
33.4K
28.5K
28.6K
30.1K
30.9K
35.8K
41.4K
46.1K
52.4K
58.3K
57.3K
52.3K
51.9K
49.5K
42.2K
29.3K



4.2 Analyzing the entire dataset from April to September 2014

4.2.0 Preliminary Steps

In [ ]:
```
##----Adding three new columns of Date,Hour and Day of the week----##

uber_data['Date'] = uber_data['datetime'].dt.date
uber_data['Hour'] = uber_data['datetime'].dt.hour
uber_data['Day_of_Week']= uber_data['datetime'].dt.day_name()
uber_data['Month'] = uber_data['datetime'].dt.month

uber_data.head()
```

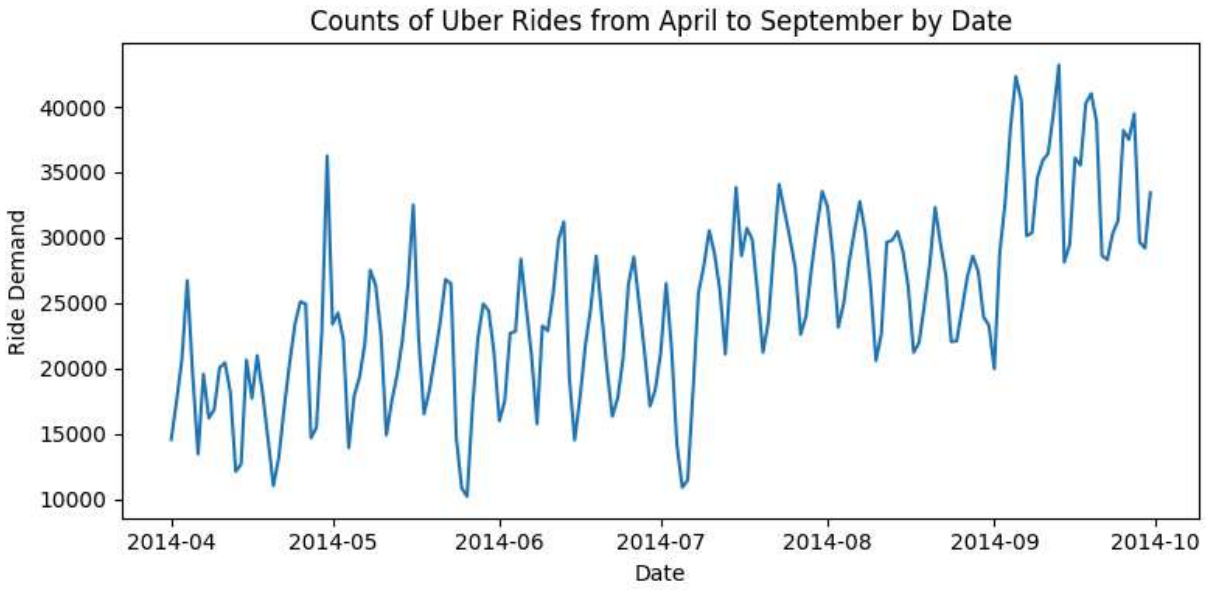
Out[ ]:

	datetime	Lat	Lon	Base	Date	Hour	Day_of_Week	Month
0	2014-04-01 00:11:00	40.7690	-73.9549	B02512	2014-04-01	0	Tuesday	4
1	2014-04-01 00:17:00	40.7267	-74.0345	B02512	2014-04-01	0	Tuesday	4
2	2014-04-01 00:21:00	40.7316	-73.9873	B02512	2014-04-01	0	Tuesday	4
3	2014-04-01 00:28:00	40.7588	-73.9776	B02512	2014-04-01	0	Tuesday	4
4	2014-04-01 00:33:00	40.7594	-73.9722	B02512	2014-04-01	0	Tuesday	4

In [ ]:
```
# Calculate counts of occurrences of Dates and Hours:
date_counts_uber = uber_data['Date'].value_counts().sort_index()
```

4.2.1 Temporal Distribution of Ride demands using Line Plot

In [ ]:
```
# Create a line plot of Dates:
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(date_counts_uber.index, date_counts_uber.values)
ax.set_xlabel('Date')
ax.set_ylabel('Ride Demand')
ax.set_title('Counts of Uber Rides from April to September by Date')

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



4.2.2 Cross Validating using Heatmaps

In [ ]:
```
def count_rows(rows):
    return len(rows)
```

```
def heatmap(col1,col2):
    by_cross=uber_data.groupby([col1,col2]).apply(count_rows)
    pivot=by_cross.unstack()
    plt.figure(figsize=(20,15));
    return px.imshow(pivot,
          labels=dict(color="Number Of Trips"),
          title=' Heatmap by {} and {}'.format(col1,col2)
         )
```

4.2.2.1 Hourly Trip Distribution Across Months

In [ ]:
```
heatmap('Month','Hour')
```

```
<ipython-input-15-f731f7429263>:5: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation. Either pass `include_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warning.
  by_cross=uber_data.groupby([col1,col2]).apply(count_rows)
```

```
<Figure size 2000x1500 with 0 Axes>
```

4.2.2.2 Hourly Trip Distribution Across Days of the Week

In [ ]:
```
heatmap('Day_of_Week','Hour')
```

```
<ipython-input-15-f731f7429263>:5: DeprecationWarning:

DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecate
d, and in a future version of pandas the grouping columns will be excluded from the
operation. Either pass `include_groups=False` to exclude the groupings or explicitly
select the grouping columns after groupby to silence this warning.
```

```
<Figure size 2000x1500 with 0 Axes>
```

4.2.2.3 Hourly Trip Distribution Across different Dates

In [ ]:
```
heatmap('Date','Hour')
```

```
<ipython-input-15-f731f7429263>:5: DeprecationWarning:

DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecate
d, and in a future version of pandas the grouping columns will be excluded from the
operation. Either pass `include_groups=False` to exclude the groupings or explicitly
select the grouping columns after groupby to silence this warning.
```

<Figure size 2000x1500 with 0 Axes>

4.3 Analyzing the entire dataset by Location

4.3.0 Preliminary step

In [ ]:
```
##-----Aggregate ride demand per hour-----##

total_demand = uber_data.groupby(["Lat", "Lon"]).size().reset_index(name="trip_coun

# Initialize a Folium map centered on NYC
#m = folium.Map(location=[40.7128, -74.0060], zoom_start=11)

# Prepare heatmap data (lat, lon, ride count)
heat_data = list(zip(total_demand["Lat"], total_demand["Lon"], total_demand["trip_c

# Add heatmap layer
#HeatMap(heat_data, radius=15, blur=10).add_to(m)

# Save & display the map
#m.save("uber_hourly_heatmap.html")
#m
```

In [ ]:
```
# Load NYC Taxi Zones shapefile (if available)
nyc_zones = gpd.read_file("/kaggle/input/taxi-zone-shape-files")
```

In [ ]:
```
nyc_zones = nyc_zones.to_crs(epsg=4326)
```

4.3.0.1 Identifying the top 3 locations with the highest ride demand

In [ ]:
```
# Aggregate ride demand per location
top_locations = uber_data.groupby(["Lat", "Lon"]).size().reset_index(name="trip_cou

# Get the top 3 locations with the highest demand
top_3 = top_locations.sort_values(by="trip_count", ascending=False).head(3)
print(top_3)
```

```
           Lat      Lon  trip_count
32881   40.6448 -73.7819        2299
432167  40.7685 -73.8625        2257
32880   40.6448 -73.7820        2079
```

In [ ]:
```
## Initialize a Folium map
#m = folium.Map(location=[40.7128, -74.0060], zoom_start=11)

# Add heatmap layer
#HeatMap(heat_data, radius=15, blur=10).add_to(m)

# Add markers for high-demand locations
#for index, row in top_3.iterrows():
#    folium.Marker(
#        location=[row["Lat"], row["Lon"]],
#        popup=f"High Demand Location: {row['trip_count']} Rides",
#        icon=folium.Icon(color="red", icon="info-sign")
#    ).add_to(m)

# Save & display the map
#m.save("uber_heatmap_with_markers.html")
#m
```

In [ ]:
```
#top_3 = top_3.drop(["geometry", "index_right","OBJECTID","Shape_Leng","Shape_Area"
```

In [ ]:
```
top_3.head()
```

Out[ ]:

	Lat	Lon	trip_count
32881	40.6448	-73.7819	2299
432167	40.7685	-73.8625	2257
32880	40.6448	-73.7820	2079

In [ ]:
```
##----Performing a spatial join to get the taxi zone names for top locations-----##

top_3_gdf = gpd.GeoDataFrame(top_3, geometry=gpd.points_from_xy(top_3["Lon"], top_3
top_3 = gpd.sjoin(top_3_gdf, nyc_zones, how="left", predicate="intersects")
```

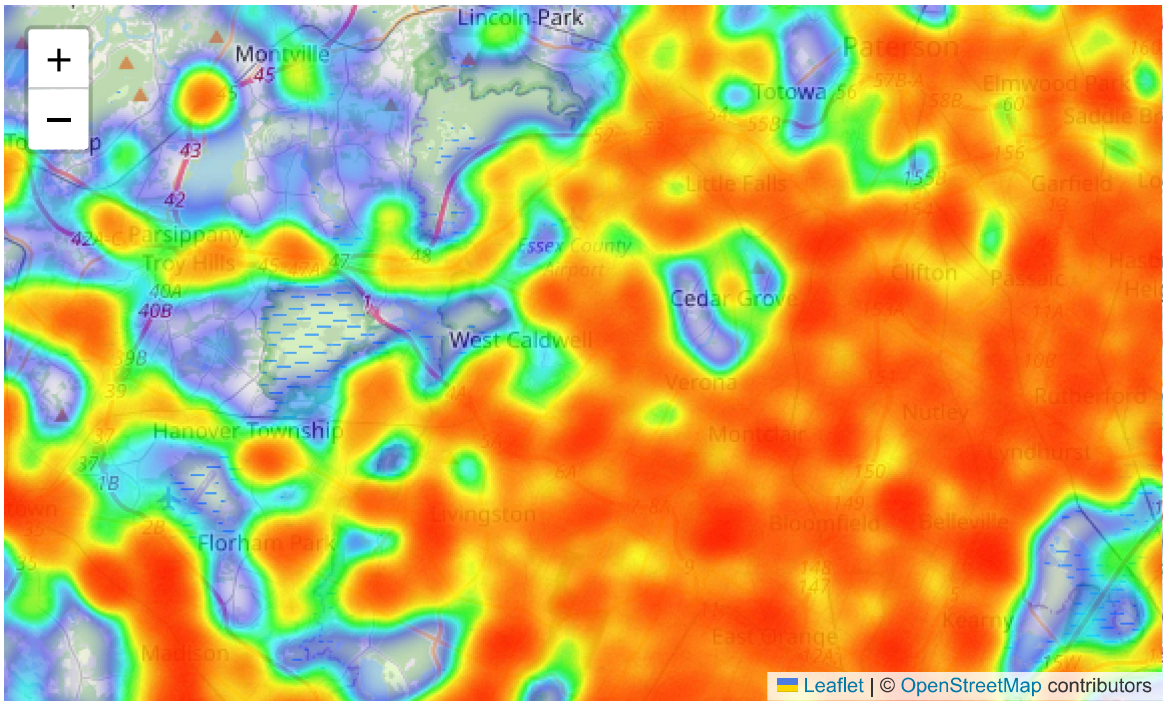
```
# Ensure top_5 now has 'zone' column (Taxi Zone Name)
print(top_3[["Lat", "Lon", "trip_count", "zone"]].head(3))
```

```
           Lat      Lon  trip_count              zone
32881   40.6448 -73.7819        2299        JFK Airport
432167  40.7685 -73.8625        2257  LaGuardia Airport
32880   40.6448 -73.7820        2079        JFK Airport
```

4.3.1 Observing the top 3 location using Geospatial Heatmap

In [ ]:
```
import folium

# Initialize a Folium map
m = folium.Map(location=[40.7128, -74.0060], zoom_start=11)

# Add heatmap layer
HeatMap(heat_data, radius=15, blur=10).add_to(m)

# Add markers with location names
for index, row in top_3.iterrows():
    folium.Marker(
        location=[row["Lat"], row["Lon"]],
        popup=f"{row['zone']}: {row['trip_count']} Rides",
        icon=folium.Icon(color="red", icon="info-sign")
    ).add_to(m)

# Save & display map
m.save("uber_heatmap_with_location_names.html")
m
```

Out[ ]:



5. Time Series Modelling

5.0 Preliminary Steps

In [ ]:
```
##-----Grouping the dataset into hourly intervals----##
uber_time_series = uber_data.resample("H", on="datetime").size().reset_index(name='
```

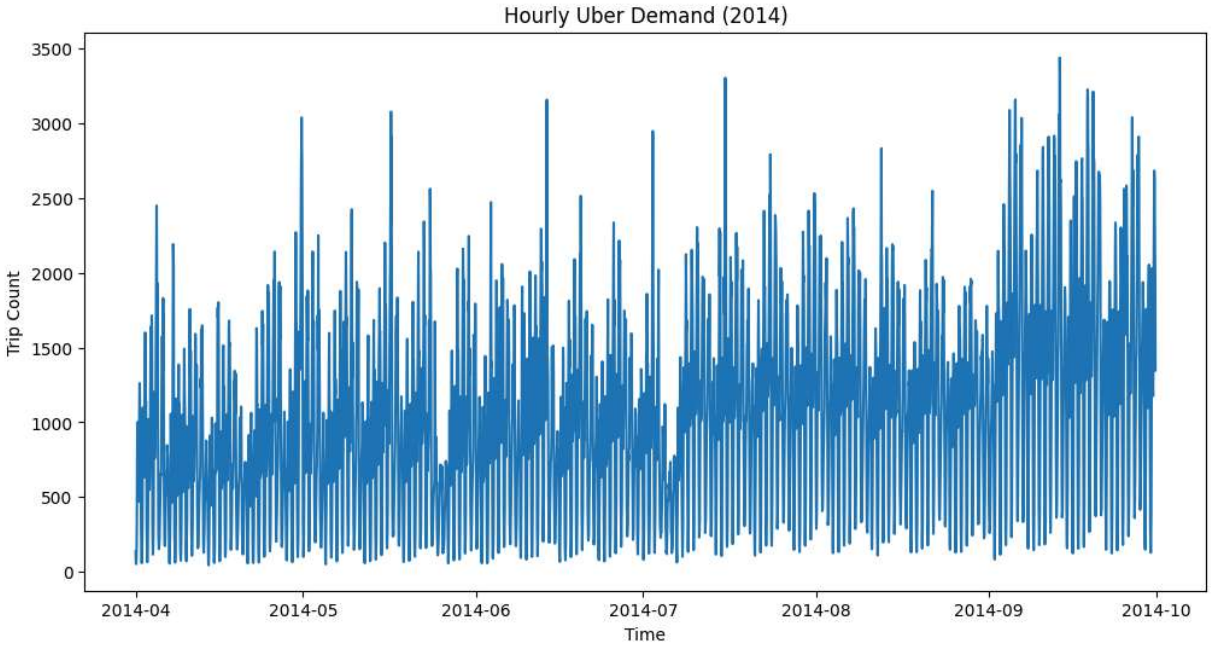
```
<ipython-input-26-6612e3c450c2>:2: FutureWarning:

'H' is deprecated and will be removed in a future version, please use 'h' instead.
```

In [ ]:
```
##-----Defining a function used for evaluation metrics-----##

def evaluate_model(y_true, y_pred, model_name="Model"):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    print(f"{model_name} Evaluation:")
    print(f"MAE: {mae:.2f}")
    print(f"MSE: {mse:.2f}")
    print(f"RMSE: {rmse:.2f}")
    print(f"MAPE: {mape:.2f}%\n")
```

5.1 Hourly Temporal Distribution of Ride Demands

In [ ]:
```
# Plot demand trends
plt.figure(figsize=(12, 6))
sns.lineplot(x=uber_time_series['datetime'], y=uber_time_series['trip_count'])
plt.title('Hourly Uber Demand (2014)')
plt.xlabel('Time')
plt.ylabel('Trip Count')
plt.show()
```

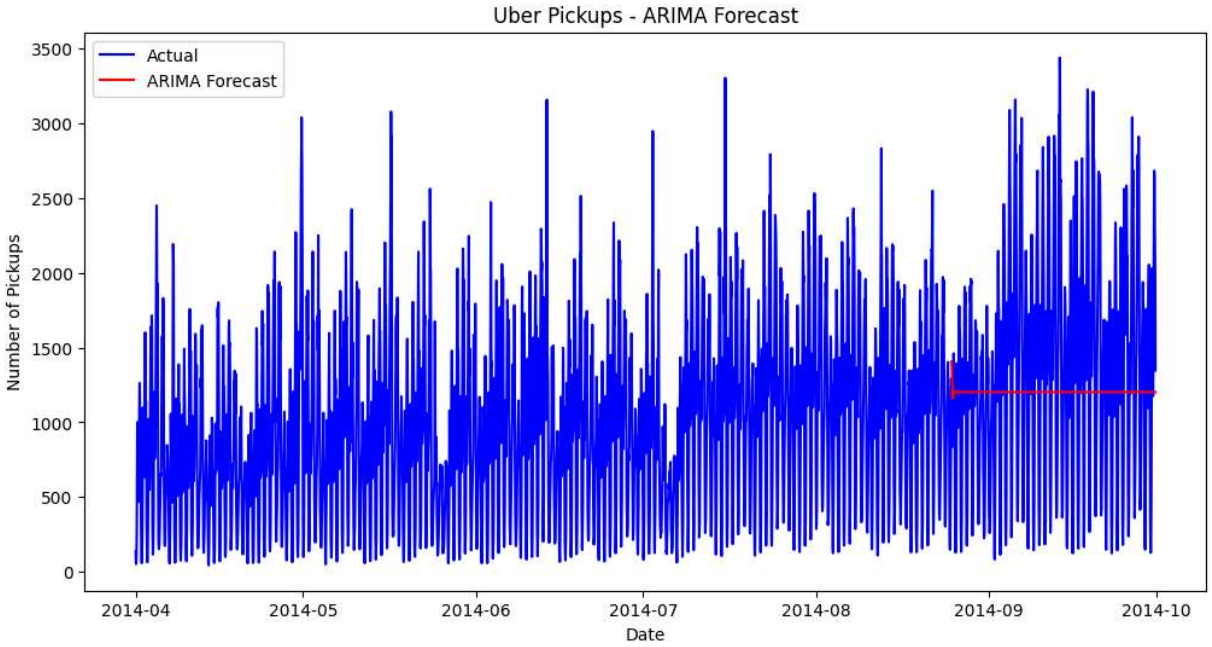
```
/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert
inf values to NaN before operating instead.

/usr/local/lib/python3.10/dist-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert
inf values to NaN before operating instead.
```



In [ ]:
```
##----Spillting dataset into training and testing used for Time series modelling---
train_size = int(len(uber_time_series) * 0.8)
train, test = uber_time_series["trip_count"][:train_size], uber_time_series["trip_c
```

5.2 Prediction using Arima Model

In [ ]:
```
arima_m = ARIMA(train, order=(5,1,0))
model_fit = arima_m.fit()

forecast = model_fit.forecast(steps=len(test))

forecast_index = uber_time_series["datetime"][train_size:train_size + len(forecast)

plt.figure(figsize=(12, 6))
plt.plot(uber_time_series["datetime"], uber_time_series["trip_count"], label="Actua
plt.plot(forecast_index, forecast, label="ARIMA Forecast", color="red")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.title("Uber Pickups - ARIMA Forecast")
plt.legend()
plt.show()
```

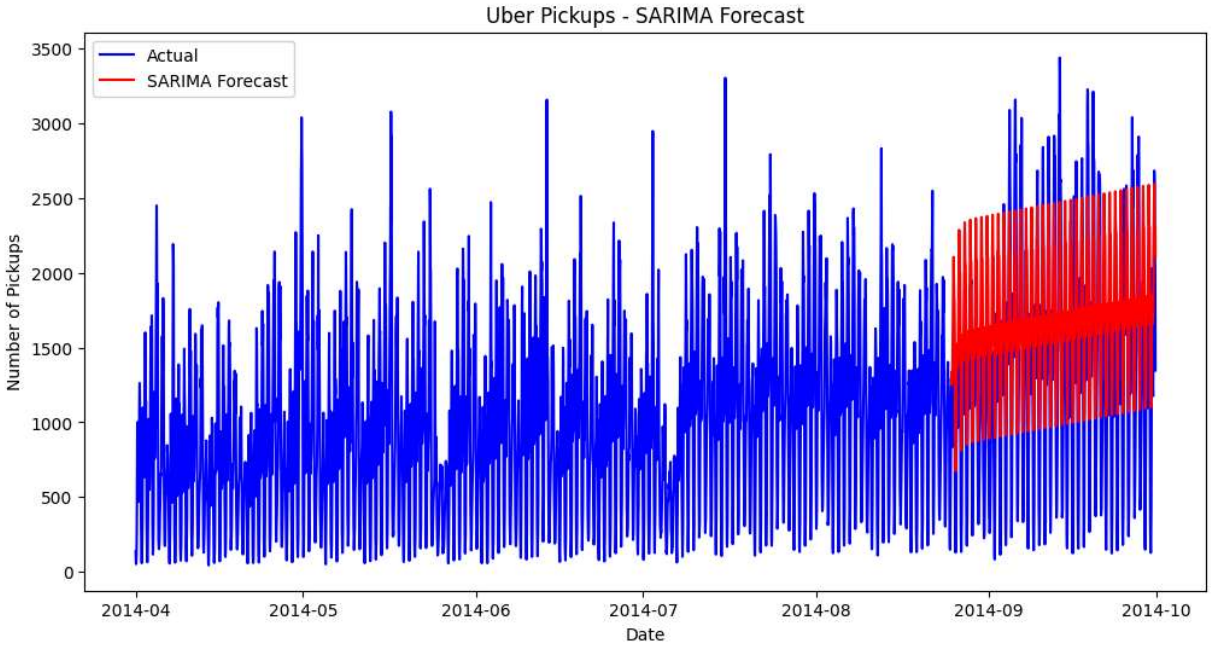


In [ ]:
```
evaluate_model(test, forecast, "ARIMA")
```

```
ARIMA Evaluation:
MAE: 623.88
MSE: 597843.26
RMSE: 773.20
MAPE: 91.84%
```

5.2 Prediction using Sarima Model

In [ ]:
```
sarima_m = SARIMAX(train, order=(1,1,1), seasonal_order=(1,1,1,24))
sarima_fit = sarima_m.fit()

forecast = sarima_fit.get_forecast(steps=len(test)).predicted_mean

forecast_index = uber_time_series["datetime"][train_size:train_size + len(forecast)

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(uber_time_series["datetime"], uber_time_series["trip_count"], label="Actua
plt.plot(forecast_index, forecast, label="SARIMA Forecast", color="red")
plt.xlabel("Date")
plt.ylabel("Number of Pickups")
plt.title("Uber Pickups - SARIMA Forecast")
plt.legend()
plt.show()
```



In [ ]:
```
evaluate_model(test, forecast, "SARIMA")
```

```
SARIMA Evaluation:
MAE: 474.58
MSE: 312986.78
RMSE: 559.45
MAPE: 81.67%
```

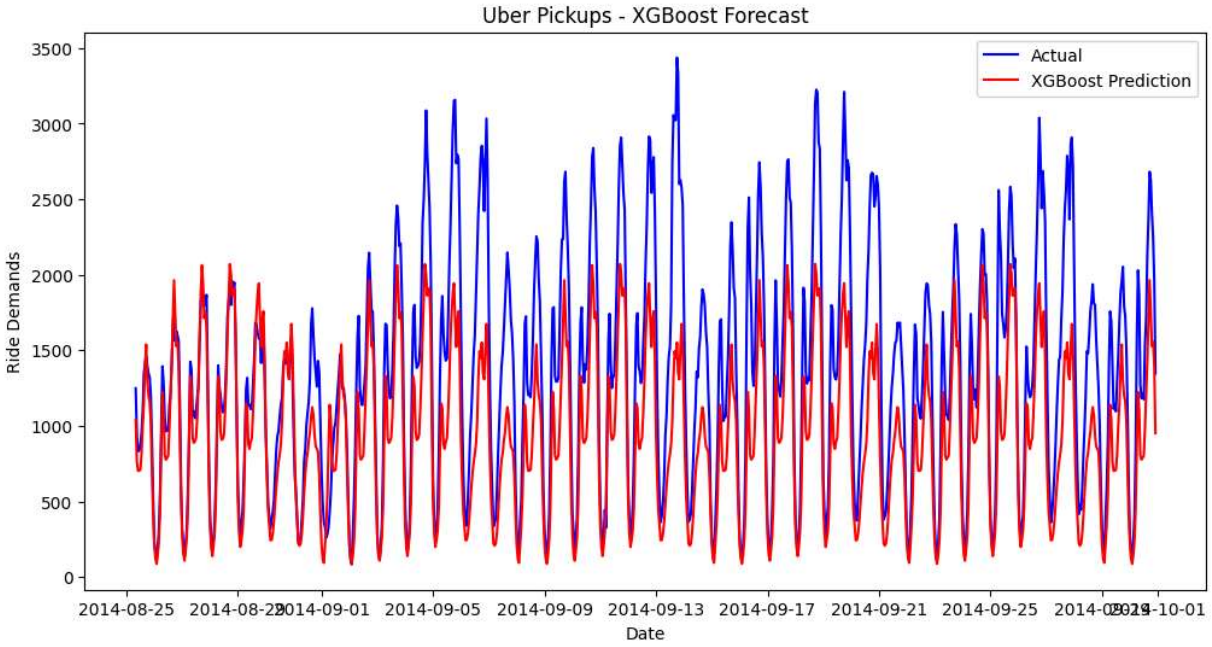
6. Machine Learning Models

6.1 Prediction using XGBoost Model

In [ ]:
```
uber_time_series["datetime"] = pd.to_datetime(uber_time_series["datetime"])
```

In [ ]:
```
# Feature Engineering
uber_time_series['Hour'] = uber_time_series['datetime'].dt.hour
uber_time_series['Day_of_Week'] = uber_time_series['datetime'].dt.weekday
uber_time_series['Date'] = uber_time_series['datetime'].dt.date  # Preserve the dat

# Set datetime as index for sequential modeling
uber_time_series.set_index("datetime", inplace=True)


# Train-Test Split
train_size = int(len(uber_time_series) * 0.8)

# Train-Test Split (Preserve Order)
X_train, X_test = uber_time_series[['Hour', 'Day_of_Week']][:train_size], uber_time
y_train, y_test = uber_time_series['trip_count'][:train_size], uber_time_series['tr

xgb = XGBRegressor(n_estimators=100, learning_rate=0.1).fit(X_train, y_train)
```

```
y_pred = xgb.predict(X_test)

plt.figure(figsize=(12, 6))
plt.plot(y_test.index, y_test, label="Actual", color="blue")
plt.plot(y_test.index, y_pred, label="XGBoost Prediction", color="red")
plt.xlabel("Date")
plt.ylabel("Ride Demands")
plt.title("Uber Pickups - XGBoost Forecast")
plt.legend()
plt.show()
```



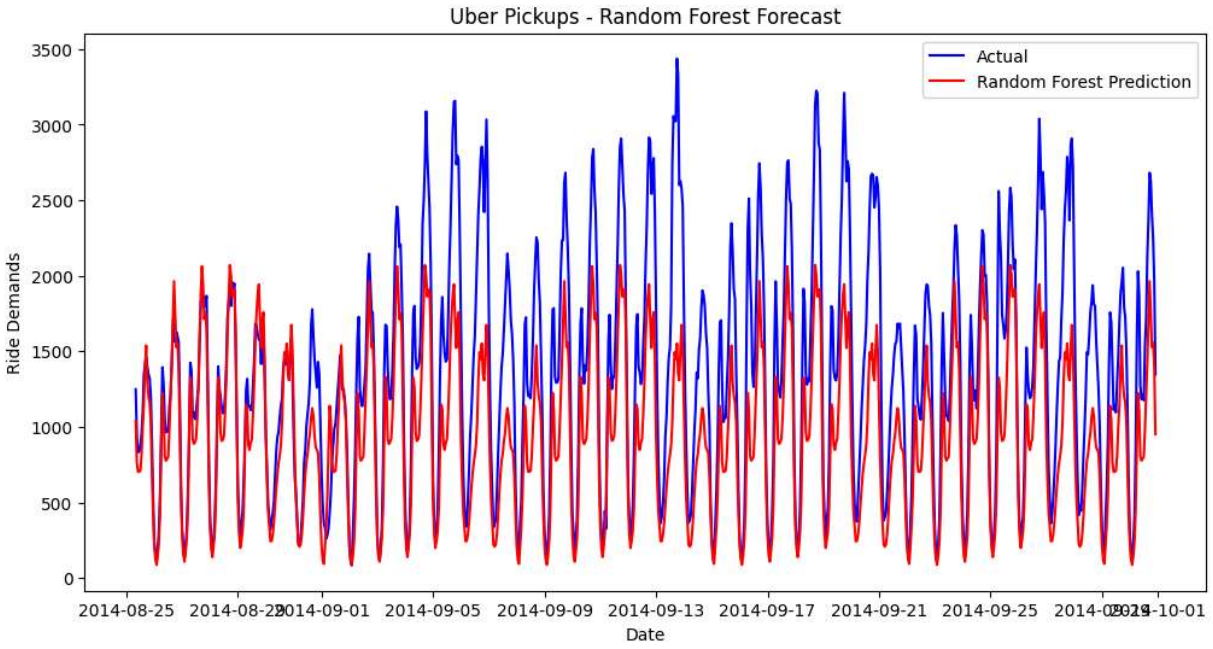
In [ ]:
```
evaluate_model(y_test, y_pred, "XGBoost")
```

```
XGBoost Evaluation:
MAE: 429.28
MSE: 296126.57
RMSE: 544.18
MAPE: 30.76%
```

6.2 Prediction using Random Forest Model

In [ ]:
```
# Feature Engineering
#uber_time_series['Hour'] = uber_time_series['datetime'].dt.hour
#uber_time_series['Day_of_Week'] = uber_time_series['datetime'].dt.weekday
#uber_time_series['Date'] = uber_time_series['datetime'].dt.date  # Preserve the da

# Set datetime as index for sequential modeling
#uber_time_series.set_index("datetime", inplace=True)


# Train-Test Split
#train_size = int(len(uber_time_series) * 0.8)

# Train-Test Split (Preserve Order)
#X_train, X_test = uber_time_series[['Hour', 'Day_of_Week']][:train_size], uber_tim
```

```
#y_train, y_test = uber_time_series['trip_count'][:train_size], uber_time_series['t


rf = RandomForestRegressor(n_estimators=100).fit(X_train, y_train)

y_pred_rf = rf.predict(X_test)

plt.figure(figsize=(12, 6))
plt.plot(y_test.index, y_test, label="Actual", color="blue")
plt.plot(y_test.index, y_pred, label="Random Forest Prediction", color="red")
plt.xlabel("Date")
plt.ylabel("Ride Demands")
plt.title("Uber Pickups - Random Forest Forecast")
plt.legend()
plt.show()
```



In [ ]:
```
evaluate_model(y_test, y_pred_rf, "Random Forest")
```

```
Random Forest Evaluation:
MAE: 429.24
MSE: 295734.17
RMSE: 543.81
MAPE: 30.76%
```

6.3 Prediction using LSTM Model

In [ ]:
```
# Initialize MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))

# Normalize the `trip_count` column
#uber_time_series['trip_count_scaled'] = scaler.fit_transform(uber_data[['trip_coun

uber_time_series["trip_count_scaled"] = scaler.fit_transform(uber_time_series["trip
# Prepare LSTM Data (Using Scaled Values)
X_lstm, y_lstm = [], []
```

```
sequence_length = 72  # Using last 72 hours

for i in range(len(uber_time_series) - sequence_length):
    X_lstm.append(uber_time_series['trip_count_scaled'][i:i+sequence_length])
    y_lstm.append(uber_time_series['trip_count_scaled'][i+sequence_length])

# Convert to numpy arrays
X_lstm, y_lstm = np.array(X_lstm), np.array(y_lstm)

# Reshape input for LSTM [samples, time steps, features]
X_lstm = np.expand_dims(X_lstm, axis=2)

# Split Data into Training & Testing Sets
train_size = int(len(X_lstm) * 0.8)
X_train, X_test = X_lstm[:train_size], X_lstm[train_size:]
y_train, y_test = y_lstm[:train_size], y_lstm[train_size:]

# Define LSTM Model
model_lstm = Sequential([
    LSTM(50, activation="relu", return_sequences=True, input_shape=(sequence_length
    Dropout(0.2),
    LSTM(50, activation="relu"),
    Dropout(0.2),
    Dense(25, activation="relu"),
    Dense(1)
])


# Compile Model
model_lstm.compile(optimizer='adam', loss='mse')

# Train LSTM Model
history = model_lstm.fit(X_train, y_train, epochs=50, batch_size=32, validation_dat

y_pred = model_lstm.predict(X_test)

y_pred_rescaled = scaler.inverse_transform(y_pred.reshape(-1, 1))
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plot Actual vs. Predicted Ride Demand
test_index = uber_time_series.index[train_size + sequence_length:]  # Ensure proper

plt.figure(figsize=(12, 6))
plt.plot(test_index, y_test_rescaled, label="Actual Ride Demand", color="blue")
plt.plot(test_index, y_pred_rescaled, label="LSTM Prediction", color="red", linesty

plt.xlabel("Date")
plt.ylabel("Number of Uber Pickups")
plt.title("Actual vs. Predicted Uber Ride Demand (LSTM Model)")
plt.legend()
plt.xticks(rotation=45)
plt.show()

# Save the trained LSTM model
model_lstm.save("lstm_trip_count_model.h5")
print("Model saved successfully!")
```

```
# Save the scaler to a file
joblib.dump(scaler, "scaler.pkl")
```

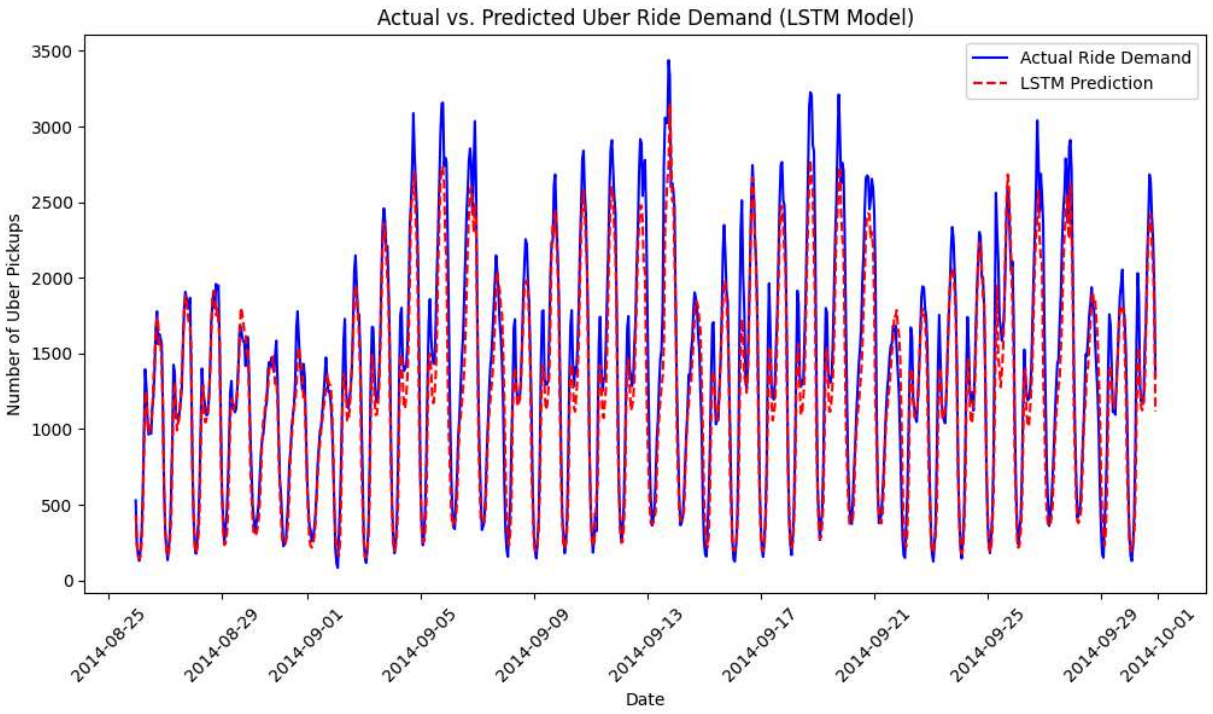
```
<ipython-input-40-c65a3565916b>:14: FutureWarning:

Series.__getitem__ treating keys as positions is deprecated. In a future version, in
teger keys will always be treated as labels (consistent with DataFrame behavior). To
access a value by position, use `ser.iloc[pos]`
```

```
Epoch 1/50
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarnin
g:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model instea
d.
```

```
108/108 ━━━━━━━━━━━━━━━━━━━━ 8s 37ms/step - loss: 0.0517 - val_loss: 0.0608
Epoch 2/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0267 - val_loss: 0.0248
Epoch 3/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0132 - val_loss: 0.0120
Epoch 4/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0092 - val_loss: 0.0120
Epoch 5/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0089 - val_loss: 0.0101
Epoch 6/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0077 - val_loss: 0.0116
Epoch 7/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0078 - val_loss: 0.0114
Epoch 8/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0065 - val_loss: 0.0081
Epoch 9/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0063 - val_loss: 0.0079
Epoch 10/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0062 - val_loss: 0.0086
Epoch 11/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0060 - val_loss: 0.0082
Epoch 12/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0056 - val_loss: 0.0078
Epoch 13/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0055 - val_loss: 0.0083
Epoch 14/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0056 - val_loss: 0.0069
Epoch 15/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0050 - val_loss: 0.0105
Epoch 16/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - loss: 0.0054 - val_loss: 0.0081
Epoch 17/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0047 - val_loss: 0.0074
Epoch 18/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0050 - val_loss: 0.0075
Epoch 19/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0047 - val_loss: 0.0077
Epoch 20/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0046 - val_loss: 0.0063
Epoch 21/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0040 - val_loss: 0.0067
Epoch 22/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0040 - val_loss: 0.0067
Epoch 23/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0044 - val_loss: 0.0056
Epoch 24/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0038 - val_loss: 0.0059
Epoch 25/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - loss: 0.0036 - val_loss: 0.0061
Epoch 26/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0035 - val_loss: 0.0060
Epoch 27/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0035 - val_loss: 0.0062
Epoch 28/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0035 - val_loss: 0.0071
Epoch 29/50
```

```
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0033 - val_loss: 0.0049
Epoch 30/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0033 - val_loss: 0.0054
Epoch 31/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0030 - val_loss: 0.0045
Epoch 32/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0033 - val_loss: 0.0070
Epoch 33/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - loss: 0.0033 - val_loss: 0.0056
Epoch 34/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - loss: 0.0027 - val_loss: 0.0048
Epoch 35/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0027 - val_loss: 0.0042
Epoch 36/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0029 - val_loss: 0.0061
Epoch 37/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0030 - val_loss: 0.0043
Epoch 38/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0029 - val_loss: 0.0041
Epoch 39/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0025 - val_loss: 0.0044
Epoch 40/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0028 - val_loss: 0.0048
Epoch 41/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0027 - val_loss: 0.0046
Epoch 42/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0027 - val_loss: 0.0062
Epoch 43/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026 - val_loss: 0.0042
Epoch 44/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026 - val_loss: 0.0058
Epoch 45/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026 - val_loss: 0.0034
Epoch 46/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0023 - val_loss: 0.0040
Epoch 47/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026 - val_loss: 0.0043
Epoch 48/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0026 - val_loss: 0.0037
Epoch 49/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0023 - val_loss: 0.0044
Epoch 50/50
108/108 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - loss: 0.0025 - val_loss: 0.0051
27/27 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step
```




```
Model saved successfully!
```

Out[ ]:
```
['scaler.pkl']
```

In [ ]:
```
evaluate_model(y_test_rescaled, y_pred_rescaled, "LSTM")
```

```
LSTM Evaluation:
MAE: 179.31
MSE: 58542.56
RMSE: 241.96
MAPE: 15.61%
```

7. Prediction of Ride demands for a specific date and time using our trained LSTM Model

In [ ]:
```
# Load the trained LSTM model with correct loss function
model_lstm = load_model("lstm_trip_count_model.h5", custom_objects={"mse": tensorfl
```

In [ ]:
```
# Load the scaler
scaler = joblib.load("scaler.pkl")
```

In [ ]:
```
# Define the target date & time
future_datetime = datetime.datetime(2025, 2, 21, 17, 0)  # February 21, 2025, at 17

# Extract the necessary features
future_hour = future_datetime.hour  # Extract hour (17)
future_day_of_week = future_datetime.weekday()  # 0=Monday, ..., 6=Sunday

# Print extracted features
```

```
print(f"Predicting for Date: {future_datetime.strftime('%Y-%m-%d %H:%M')}")
print(f"Hour: {future_hour}, Day of Week: {future_day_of_week}")
```

```
Predicting for Date: 2025-02-21 17:00
Hour: 17, Day of Week: 4
```

In [ ]:
```
# Extract only past 72 ride demand values
past_72_hours = uber_time_series["trip_count_scaled"].values[-72:]

# Extract time-based features
future_hour = 17  # Example: 5 PM
future_day_of_week = 4  # Example: Friday

# Stack all features into correct shape
future_input = np.hstack((past_72_hours.reshape(-1, 1), np.tile([future_hour, futur

# Reshape for LSTM input
future_input = future_input.reshape(1, 72, 3)  #(batch_size, time_steps, features)

print(f"Future Input Shape: {future_input.shape}")
```

```
Future Input Shape: (1, 72, 3)
```

In [ ]:
```
# Extract only past 72 ride demand values
past_72_hours = uber_time_series["trip_count_scaled"].values[-72:]  # Ensure 72 val

# Reshape to match LSTM input expectations
future_input = past_72_hours.reshape(1, 72, 1)  # (batch_size, time_steps, features

# Predict scaled ride demand
predicted_scaled_demand = model_lstm.predict(future_input)

# Convert back to actual trip count
predicted_demand = scaler.inverse_transform(predicted_scaled_demand.reshape(-1, 1))

print(f"Predicted Uber Ride Demand: {int(predicted_demand)} rides")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 429ms/step
Predicted Uber Ride Demand: 623 rides
```

8. Conclusion

-- This project successfully analyzed the Uber ride demand patterns in New York City using historical data from 2014. By leveraging time-series forecasting, geospatial mapping, and machine learning models (ARIMA, SARIMA, XGBoost, LSTM, and Random Forest), we gained key insights into ride demand trends, peak hours, and high-demand locations.

-- Based on evaluation metrics (Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE)), LSTM outperformed other models in forecasting accuracy. We used the trained LSTM model to predict ride demand for a specific date and time, demonstrating its practical application.

AAI 530 Mod 7 Assignment

Sanjay Kumar

Feb 2025

In [15]:
```
# Github link: https://github.com/sanjaykr33/SK-AAI530-PRJ/blob/master/sk_AA1530_Ub
```

In [16]:
```
# Tableau link:
```

---------- Uber dataset analysis ----------

In [17]:
```
# Import libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler




# Define file paths
uber_files = [
    "/content/sample_data/uber-raw-data-apr14.csv", "/content/sample_data/uber-raw-
    "/content/sample_data/uber-raw-data-jun14.csv", "/content/sample_data/uber-raw-
    "/content/sample_data/uber-raw-data-aug14.csv", "/content/sample_data/uber-raw-
]

# Merge dataset
df_list = [pd.read_csv(file) for file in uber_files]
df = pd.concat(df_list, ignore_index=True)
merged_file_path = "/content/sample_data/uber-raw-data-aprsep-14.csv"
df.to_csv(merged_file_path, index=False)

# Read merged dataset
df = pd.read_csv(merged_file_path)

# Inspect dataset
print("First 5 rows:")
print(df.head())
print("\nData types:")
print(df.dtypes)
```

```
print("\nDataset statistics:")
print(df.describe())

# Convert Date/Time to datetime
df['Date/Time'] = pd.to_datetime(df['Date/Time'], errors='coerce')
print("\nUpdated Data Types:")
print(df.dtypes)

# Create a copy for cleaning
df_clean = df.copy()

# Explore dataset
print("\nMissing values:")
print(df_clean.isnull().sum())
print("\nDuplicates:", df_clean.duplicated().sum())

# Clean dataset
# Drop rows with missing values
df_clean.dropna(inplace=True)
# Remove duplicates
df_clean.drop_duplicates(inplace=True)

# Save clean dataset
clean_file_path = "/content/sample_data/uber-raw-data-aprsep-14-clean.csv"
df_clean.to_csv(clean_file_path, index=False)

print("\nClean dataset saved successfully!")

# Inspect clean dataset
print("First 5 rows:")
print(df_clean.head())
print("\nData types:")
print(df_clean.dtypes)
print("\nDataset statistics:")
print(df_clean.describe())
```

```
First 5 rows:
          Date/Time      Lat      Lon    Base
0  4/1/2014 0:11:00  40.7690 -73.9549  B02512
1  4/1/2014 0:17:00  40.7267 -74.0345  B02512
2  4/1/2014 0:21:00  40.7316 -73.9873  B02512
3  4/1/2014 0:28:00  40.7588 -73.9776  B02512
4  4/1/2014 0:33:00  40.7594 -73.9722  B02512

Data types:
Date/Time     object
Lat          float64
Lon          float64
Base          object
dtype: object

Dataset statistics:
                Lat           Lon
count  4.534327e+06  4.534327e+06
mean   4.073926e+01 -7.397302e+01
std    3.994991e-02  5.726670e-02
min    3.965690e+01 -7.492900e+01
25%    4.072110e+01 -7.399650e+01
50%    4.074220e+01 -7.398340e+01
75%    4.076100e+01 -7.396530e+01
max    4.211660e+01 -7.206660e+01

Updated Data Types:
Date/Time    datetime64[ns]
Lat                 float64
Lon                 float64
Base                 object
dtype: object

Missing values:
Date/Time    0
Lat          0
Lon          0
Base         0
dtype: int64

Duplicates: 82581

Clean dataset saved successfully!
First 5 rows:
            Date/Time      Lat      Lon    Base
0 2014-04-01 00:11:00  40.7690 -73.9549  B02512
1 2014-04-01 00:17:00  40.7267 -74.0345  B02512
2 2014-04-01 00:21:00  40.7316 -73.9873  B02512
3 2014-04-01 00:28:00  40.7588 -73.9776  B02512
4 2014-04-01 00:33:00  40.7594 -73.9722  B02512

Data types:
Date/Time    datetime64[ns]
Lat                 float64
Lon                 float64
Base                 object
```

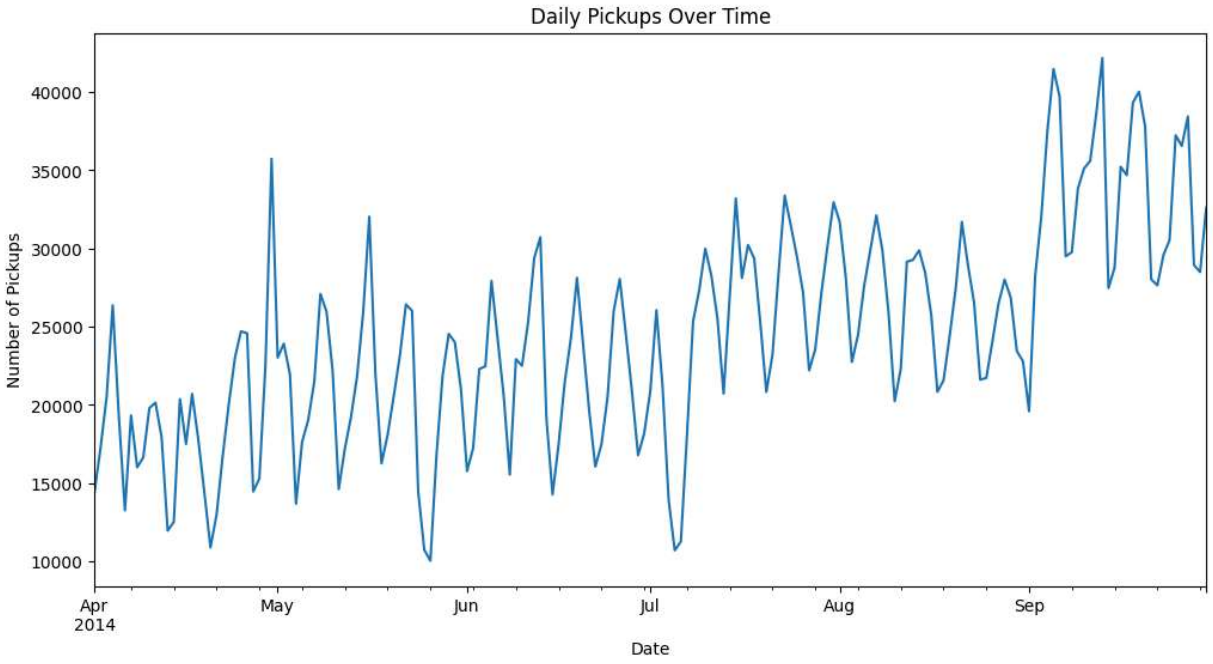
```
dtype: object

Dataset statistics:
                           Date/Time           Lat           Lon
count                        4451746  4.451746e+06  4.451746e+06
mean   2014-07-11 14:44:15.403251200  4.073924e+01 -7.397304e+01
min              2014-04-01 00:00:00  3.965690e+01 -7.492900e+01
25%              2014-05-28 10:32:00  4.072110e+01 -7.399650e+01
50%              2014-07-17 09:28:00  4.074220e+01 -7.398340e+01
75%              2014-08-27 18:47:00  4.076100e+01 -7.396530e+01
max              2014-09-30 22:59:00  4.211660e+01 -7.206660e+01
std                              NaN  3.984343e-02  5.721967e-02
```

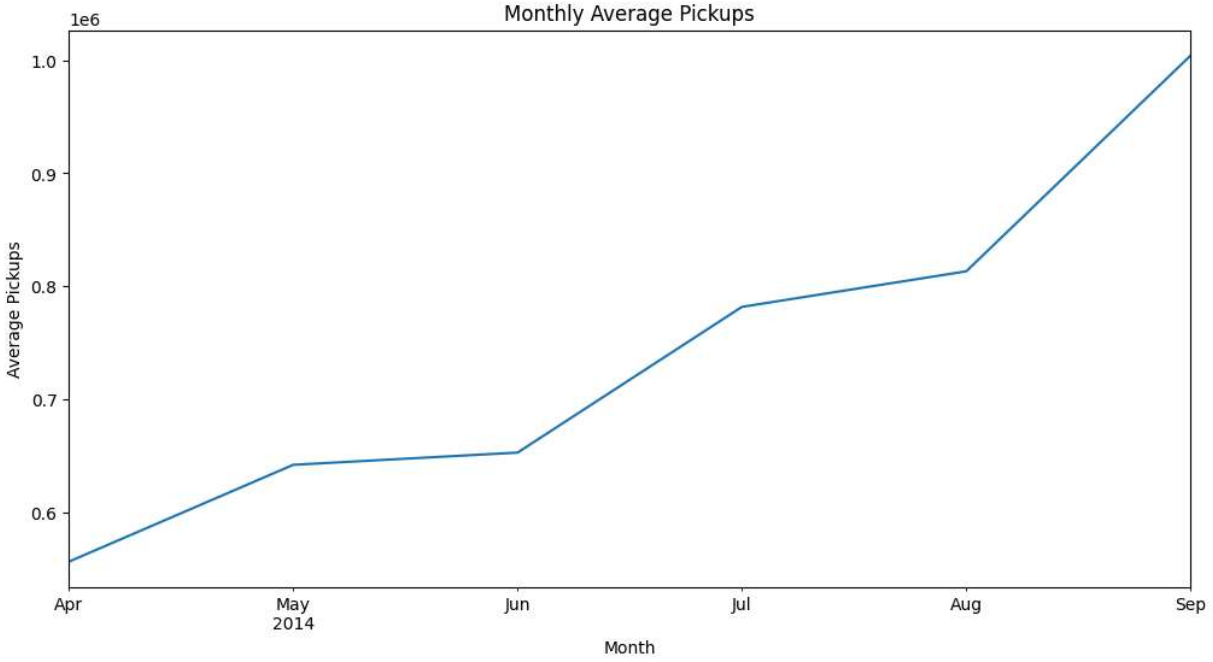
---------- End of Uber data analysis and cleanup-------

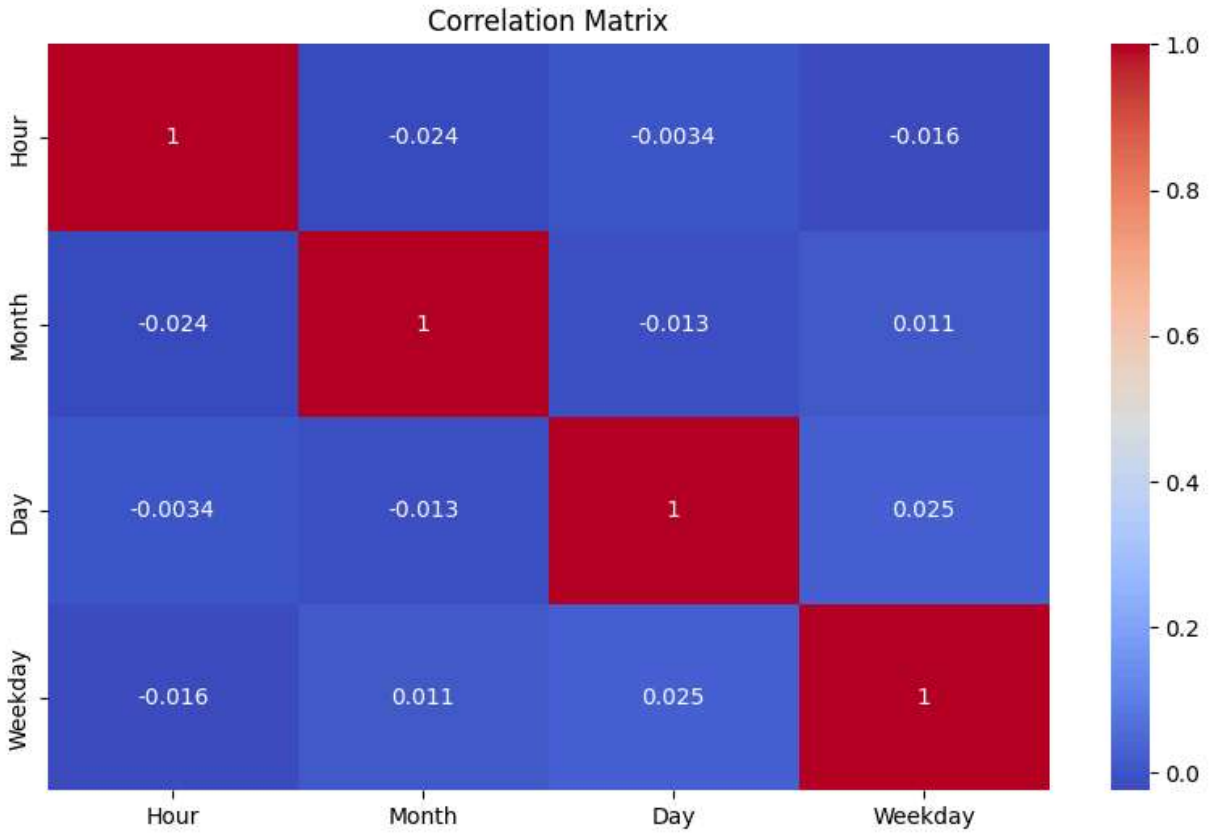
---------- Time Series Analysis ----------

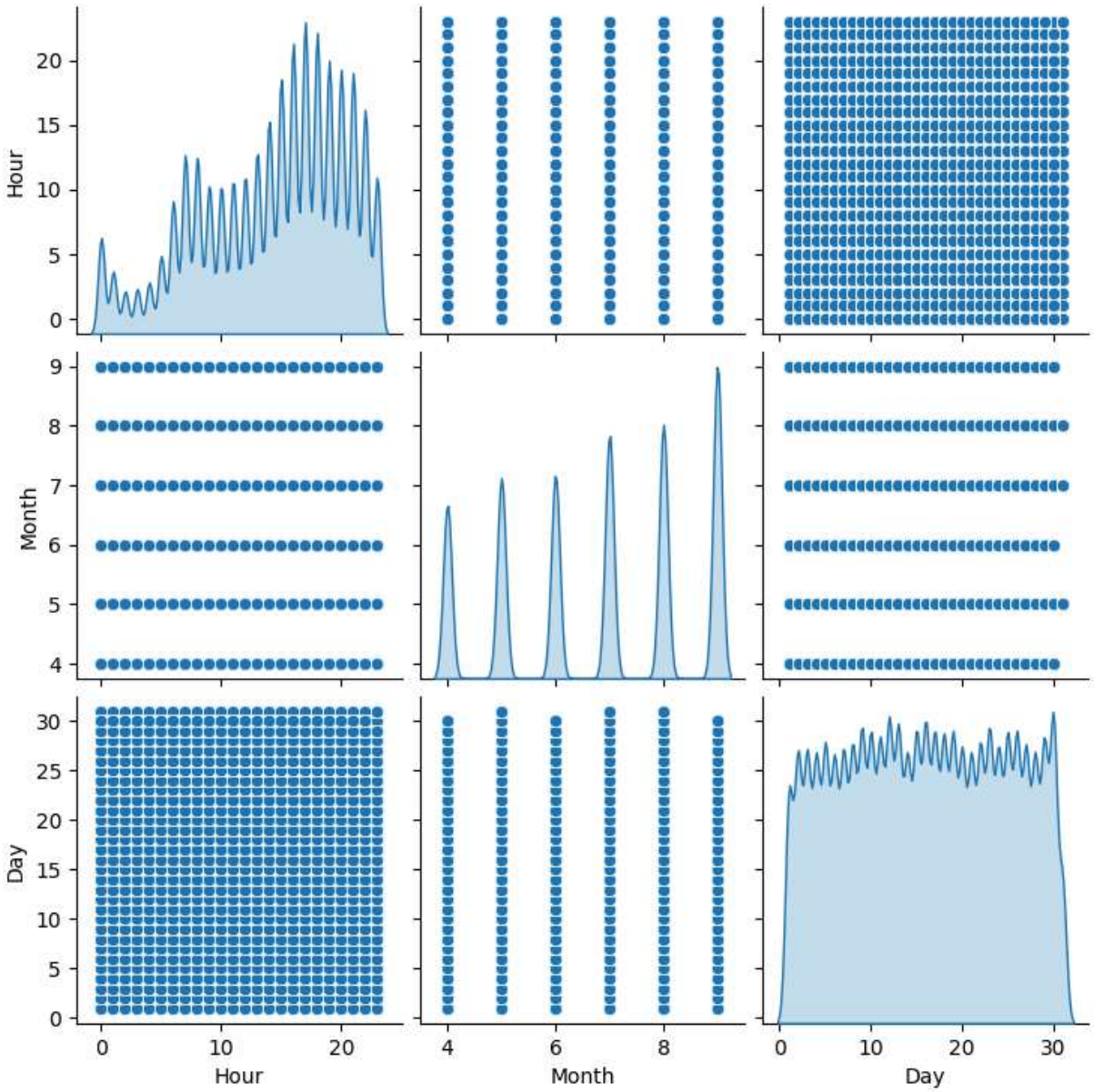
In [18]:
```
# ---------- Time Series Analysis ----------
#import matplotlib.pyplot as plt
#import seaborn as sns

# Extract time-based features
df_clean['Year'] = df_clean['Date/Time'].dt.year
df_clean['Month'] = df_clean['Date/Time'].dt.month
df_clean['Day'] = df_clean['Date/Time'].dt.day
df_clean['Hour'] = df_clean['Date/Time'].dt.hour
df_clean['Weekday'] = df_clean['Date/Time'].dt.weekday
df_clean['Weekend'] = df_clean['Weekday'] >= 5


# Visualize data over time - Daily Pickups Over Time
plt.figure(figsize=(12, 6))
df_clean.resample('D', on='Date/Time').size().plot()
plt.title('Daily Pickups Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Pickups')
plt.show()

# Compute monthly averages
monthly_avg = df_clean.resample('ME', on='Date/Time').size()

# Visualize monthly averages - Monthly Average Pickups
plt.figure(figsize=(12, 6))
monthly_avg.plot()
plt.title('Monthly Average Pickups')
plt.xlabel('Month')
plt.ylabel('Average Pickups')
plt.show()

# Correlation matrix
plt.figure(figsize=(10, 6))
sns.heatmap(df_clean[['Hour', 'Month', 'Day', 'Weekday']].corr(), annot=True, cmap=
```

```
plt.title('Correlation Matrix')
plt.show()

# Scatter plots
sns.pairplot(df_clean[['Hour', 'Month', 'Day']], diag_kind='kde')
plt.show()
```





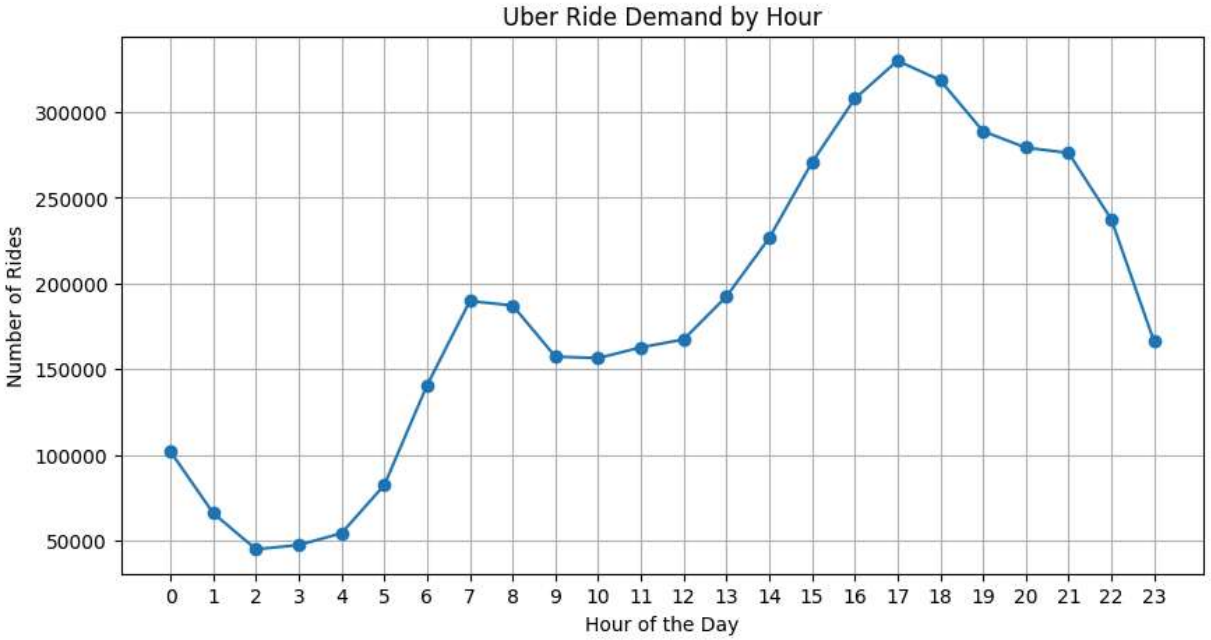

Correlation Matrix
Hour
Month
Day
Weekday
1
-0.024
-0.0034
-0.016
-0.024
1
-0.013
0.011
-0.0034
-0.013
1
0.025
-0.016
0.011
0.025
1
Hour
Month
Day
Weekday
1.0
0.8
0.6
0.4
0.2
0.0



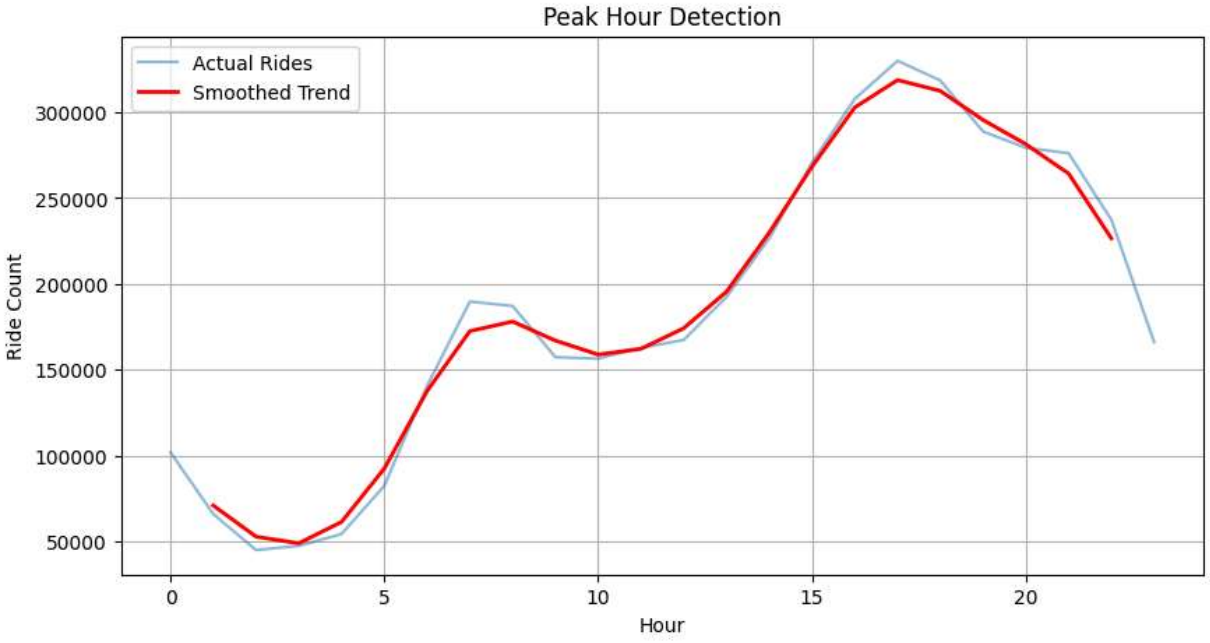
----------Peak hour Analysis ----------

In [19]:
```
#------------------ Identify Peak hours ----------------
#import pandas as pd
#import matplotlib.pyplot as plt

# Load dataset
df_clean = pd.read_csv("/content/sample_data/uber-raw-data-aprsep-14-clean.csv")

# Convert Date/Time column to datetime format
df_clean['Date/Time'] = pd.to_datetime(df_clean['Date/Time'])

# Extract hour from Date/Time
df_clean['Hour'] = df_clean['Date/Time'].dt.hour
```

```
# Aggregate ride counts by hour
hourly_rides = df_clean.groupby('Hour').size()

# Plot the ride demand per hour
plt.figure(figsize=(10, 5))
plt.plot(hourly_rides.index, hourly_rides.values, marker='o', linestyle='-')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Rides')
plt.title('Uber Ride Demand by Hour')
plt.xticks(range(0, 24))  # Ensure all hours are displayed
plt.grid()
plt.show()
```



In [20]:
```
# Apply a rolling average to smooth fluctuations
hourly_rides_smoothed = hourly_rides.rolling(window=3, center=True).mean()

# Plot
plt.figure(figsize=(10, 5))
plt.plot(hourly_rides.index, hourly_rides.values, label='Actual Rides', alpha=0.5)
plt.plot(hourly_rides.index, hourly_rides_smoothed, label='Smoothed Trend', color='
plt.xlabel('Hour')
plt.ylabel('Ride Count')
plt.legend()
plt.title('Peak Hour Detection')
plt.grid()
plt.show()
```



In [21]:
```
#import numpy as np

# Compute threshold for peak hours
threshold = hourly_rides.mean() + hourly_rides.std()

# Identify peak hours
peak_hours = hourly_rides[hourly_rides > threshold].index.tolist()
print("Identified Peak Hours:", peak_hours)
```

```
Identified Peak Hours: [16, 17, 18, 19, 20, 21]
```

----------End of Identified Peak Hours ----------

---------- Random Forest Regressor Model----

In [22]:
```
#--------------- Random Forest Regressor----
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Load cleaned dataset
df_clean = pd.read_csv("/content/sample_data/uber-raw-data-aprsep-14-clean.csv")
df_clean['Date/Time'] = pd.to_datetime(df_clean['Date/Time'])

# Extract features
df_clean['Year'] = df_clean['Date/Time'].dt.year
df_clean['Month'] = df_clean['Date/Time'].dt.month
df_clean['Day'] = df_clean['Date/Time'].dt.day
```

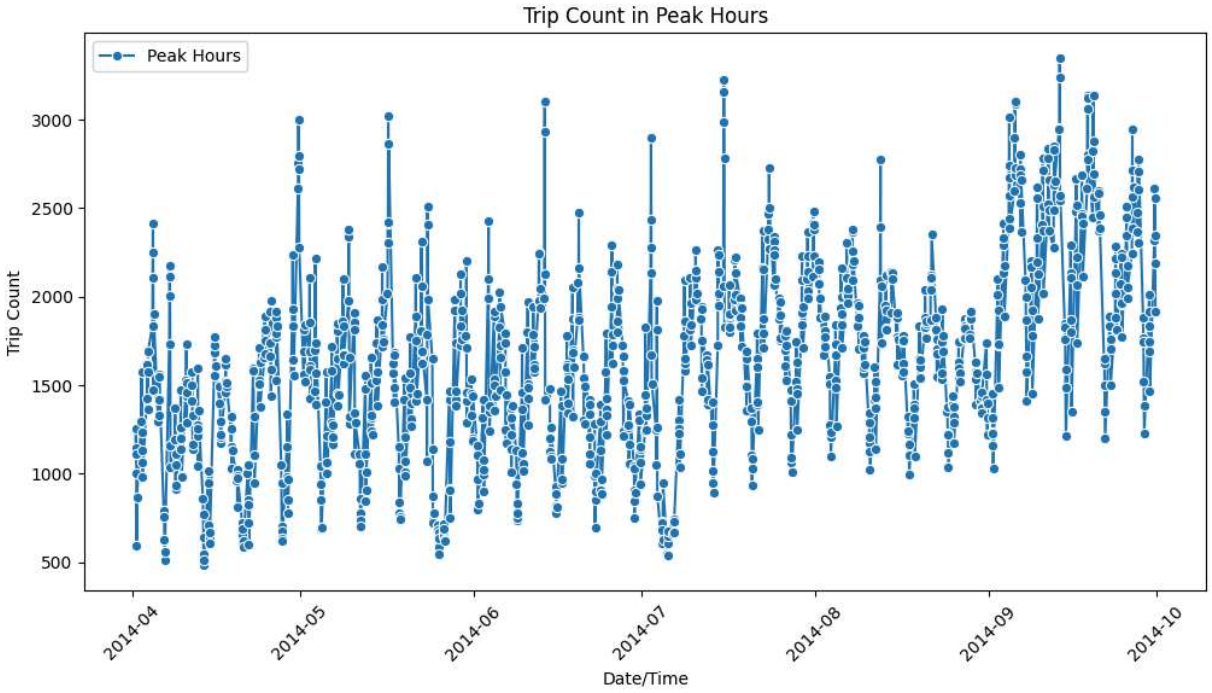
```
df_clean['Hour'] = df_clean['Date/Time'].dt.hour
df_clean['Weekday'] = df_clean['Date/Time'].dt.weekday
df_clean['Weekend'] = df_clean['Weekday'] >= 5

# Extract Longitude and Latitude
if 'Lon' in df_clean.columns and 'Lat' in df_clean.columns:
    df_clean['Lon'] = df_clean['Lon']
    df_clean['Lat'] = df_clean['Lat']

# Define target variable (trip count per hour)
df_clean['Trip_Count'] = df_clean.groupby(['Year', 'Month', 'Day', 'Hour'])['Date/T

# Drop duplicate time-based entries
df_clean = df_clean.drop_duplicates(subset=['Year', 'Month', 'Day', 'Hour'])

# ---------------- PEAK HOURS ANALYSIS -------------
# Compute threshold for peak hours
hourly_rides = df_clean.groupby('Hour')['Trip_Count'].sum()
threshold = hourly_rides.mean() + hourly_rides.std()

# Identify peak hours
peak_hours = hourly_rides[hourly_rides > threshold].index.tolist()
print("Identified Peak Hours:", peak_hours)

# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Plot trip count in peak hours
plt.figure(figsize=(12, 6))
sns.lineplot(data=peak_hour_data, x='Date/Time', y='Trip_Count', marker='o', label=
plt.xlabel("Date/Time")
plt.ylabel("Trip Count")
plt.title("Trip Count in Peak Hours")
plt.xticks(rotation=45)
plt.legend()
plt.show()

#------------
# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Group by Month and sum trip counts for peak hours
peak_hour_monthly = peak_hour_data.groupby(['Month', 'Hour'])['Trip_Count'].sum().r

# Plot trip count in peak hours per month
plt.figure(figsize=(12, 6))
sns.lineplot(data=peak_hour_monthly, x='Month', y='Trip_Count', hue='Hour', marker=
plt.xlabel("Month")
plt.ylabel("Trip Count in Peak Hours")
plt.title("Trip Count in Peak Hours Per Month")
plt.xticks(ticks=np.arange(1, 13), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'
plt.legend(title="Hour", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```

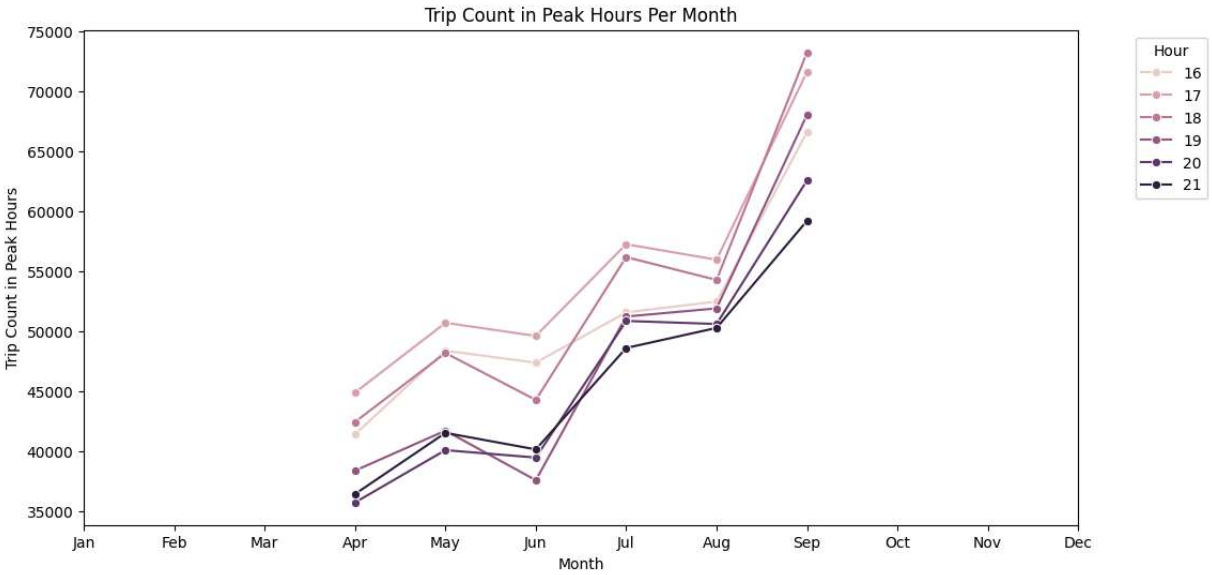
```
#---------
# Add a 'Week' column based on the Date/Time
df_clean['Week'] = df_clean['Date/Time'].dt.isocalendar().week

# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Group by Week and sum trip counts for peak hours
peak_hour_weekly = peak_hour_data.groupby(['Week', 'Hour'])['Trip_Count'].sum().res

# Plot trip count in peak hours per week
plt.figure(figsize=(12, 6))
sns.lineplot(data=peak_hour_weekly, x='Week', y='Trip_Count', hue='Hour', marker='o
plt.xlabel("Week")
plt.ylabel("Trip Count in Peak Hours")
plt.title("Trip Count in Peak Hours Per Week")
plt.legend(title="Hour", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xticks(rotation=45)
plt.show()


#---------

# Add a 'Day' column based on the Date/Time
df_clean['Day'] = df_clean['Date/Time'].dt.dayofyear

# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Group by Day and sum trip counts for peak hours
peak_hour_daily = peak_hour_data.groupby(['Day', 'Hour'])['Trip_Count'].sum().reset

# Plot trip count in peak hours per day
plt.figure(figsize=(12, 6))
sns.lineplot(data=peak_hour_daily, x='Day', y='Trip_Count', hue='Hour', marker='o')
plt.xlabel("Day of Year")
plt.ylabel("Trip Count in Peak Hours")
plt.title("Trip Count in Peak Hours Per Day")
plt.legend(title="Hour", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xticks(rotation=45)
plt.show()


#----------
# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Group by Hour and sum trip counts for peak hours
peak_hour_hourly = peak_hour_data.groupby('Hour')['Trip_Count'].sum().reset_index()

# Plot trip count in peak hours per hour
plt.figure(figsize=(12, 6))
sns.barplot(data=peak_hour_hourly, x='Hour', y='Trip_Count', palette='viridis')
plt.xlabel("Hour of Day")
plt.ylabel("Total Trip Count in Peak Hours")
plt.title("Trip Count in Peak Hours Per Hour")
```

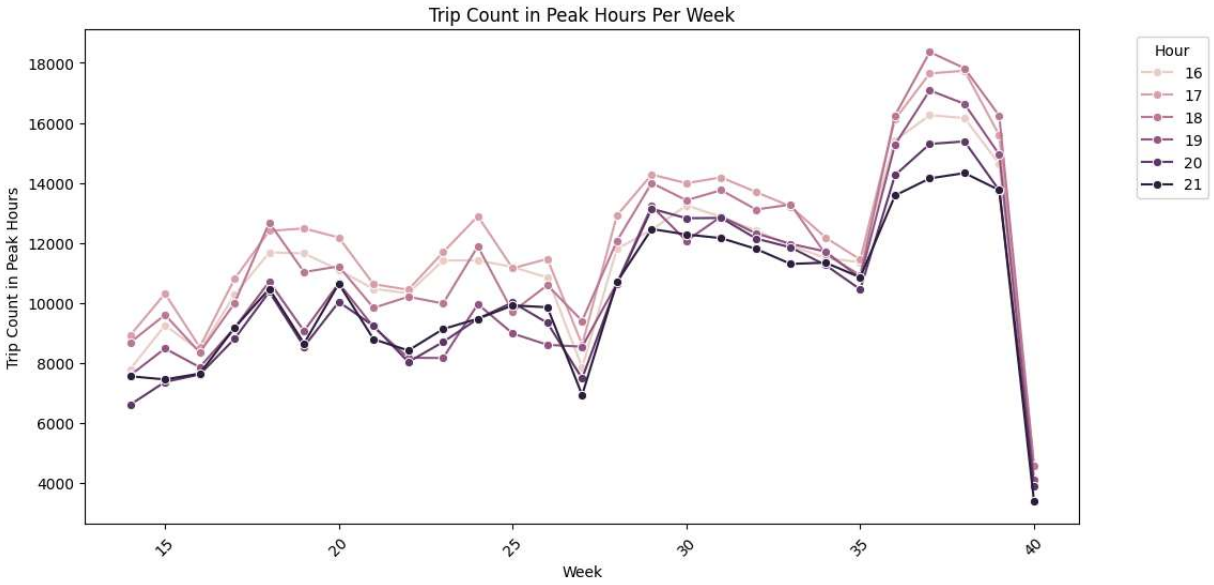
```
plt.xticks(rotation=45)
plt.show()


#----------- Train Model for Peaks hours --------
# Filter dataset for trips during peak hours
peak_hour_data = df_clean[df_clean['Hour'].isin(peak_hours)]

# Define features and target for peak hour rides prediction
features = ['Year', 'Month', 'Day', 'Hour', 'Weekday', 'Weekend', 'Lon', 'Lat']
target = 'Trip_Count'

X_peak = peak_hour_data[features]
y_peak = peak_hour_data[target]

# Split data into train and test sets for peak hour prediction
X_train_peak, X_test_peak, y_train_peak, y_test_peak = train_test_split(X_peak, y_p

# Train RandomForestRegressor for peak hour rides prediction
rf_model_peak = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model_peak.fit(X_train_peak, y_train_peak)

# Make predictions for peak hour rides
y_pred_peak = rf_model_peak.predict(X_test_peak)

# Save actual and predicted values for peak hour rides in a DataFrame
results_peak_df = pd.DataFrame({'Actual': y_test_peak, 'Predicted': y_pred_peak})
results_peak_df.to_csv("/content/sample_data/uber_rf_peak_hour_predictions.csv", in

# Plot actual vs predicted peak hour rides
plt.figure(figsize=(10, 5))
sns.scatterplot(x=y_test_peak, y=y_pred_peak, alpha=0.6)
plt.xlabel("Actual Peak Hour Rides")
plt.ylabel("Predicted Peak Hour Rides")
plt.title("Actual vs Predicted Peak Hour Rides")
plt.show()

# Calculate RMSE for peak hour rides prediction
print('RMSE for peak hour rides -------------')
rmse_peak = np.sqrt(mean_squared_error(y_test_peak, y_pred_peak))
print(f'RMSE for Peak Hour Rides Prediction: {rmse_peak:.2f}')

#-------------- Peak hour trips to Latitude ,Longitude ---------
print('Peak hour trips to Latitude , Longitude ----------------------')

plt.figure(figsize=(10, 6))
sns.scatterplot(data=peak_hour_data, x='Lon', y='Lat', alpha=0.5, s=10)
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.title("Peak Hour Trip Locations")
plt.show()

plt.figure(figsize=(10, 6))
sns.kdeplot(
    data=peak_hour_data, x='Lon', y='Lat', cmap='Reds', fill=True, thresh=0, levels
)
```

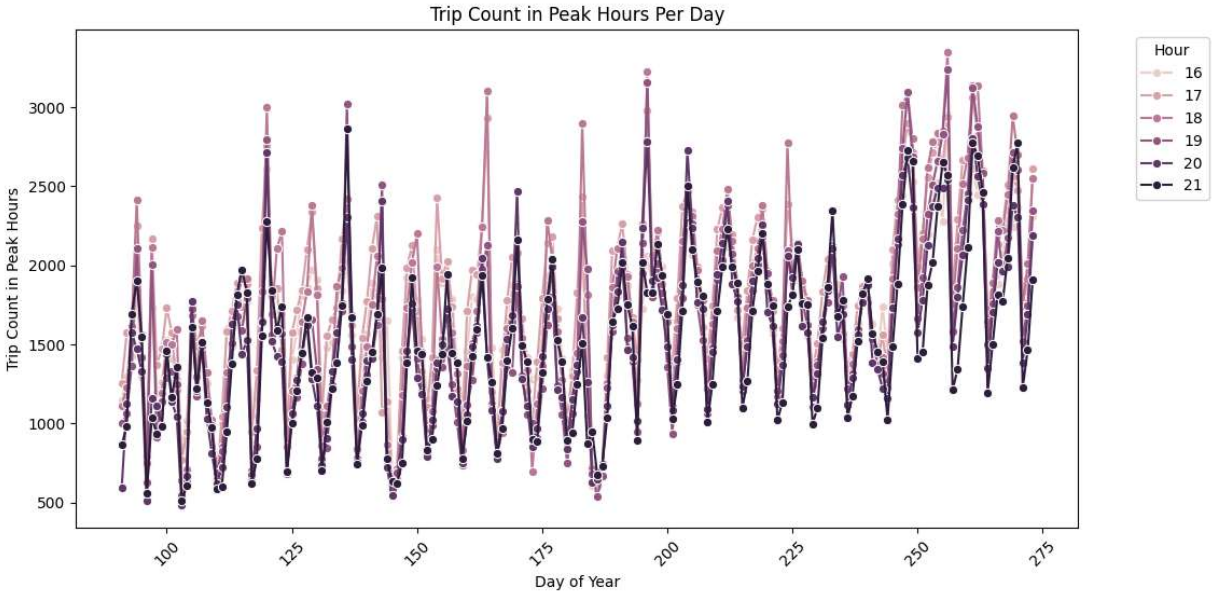
```
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.title("Density Heatmap of Peak Hour Trips")
plt.show()


#------------------- TRIP ANALYSIS FOR ALL HOURS---------------------------------

print('Trip analysis for all hours ----------------------------------')
print('-----------------------------------------------------')

# Select features and target for model
features = ['Year', 'Month', 'Day', 'Hour', 'Weekday', 'Weekend', 'Lon', 'Lat']
target = 'Trip_Count'

X = df_clean[features]
y = df_clean[target]

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Train RandomForestRegressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
y_pred = rf_model.predict(X_test)

# Save actual and predicted values in a DataFrame
results_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
results_df.to_csv("/content/sample_data/uber_rf_predictions.csv", index=False)

# Calculate RMSE
print('RMSE for all hour rides : ')
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'RMSE: {rmse:.2f}')

# Plot actual vs predicted values
plt.figure(figsize=(10, 5))
sns.scatterplot(x=y_test, y=y_pred, alpha=0.5)
plt.xlabel("Actual Trip Count")
plt.ylabel("Predicted Trip Count")
plt.title("Actual vs Predicted Trip Counts")
plt.show()

# Moving Average by Month
df_clean['Rolling_Avg'] = df_clean.groupby('Month')['Trip_Count'].transform(lambda

plt.figure(figsize=(12, 6))
sns.lineplot(x=df_clean['Month'], y=df_clean['Rolling_Avg'], marker='o')
plt.xlabel("Month")
plt.ylabel("Moving Average of Trip Count")
plt.title("Moving Average of Trip Counts by Month")
plt.show()

#------------------- END OF TRIP ANALYSIS FOR ALL HOURS----------------------------
```

```
Identified Peak Hours: [16, 17, 18, 19, 20, 21]
```

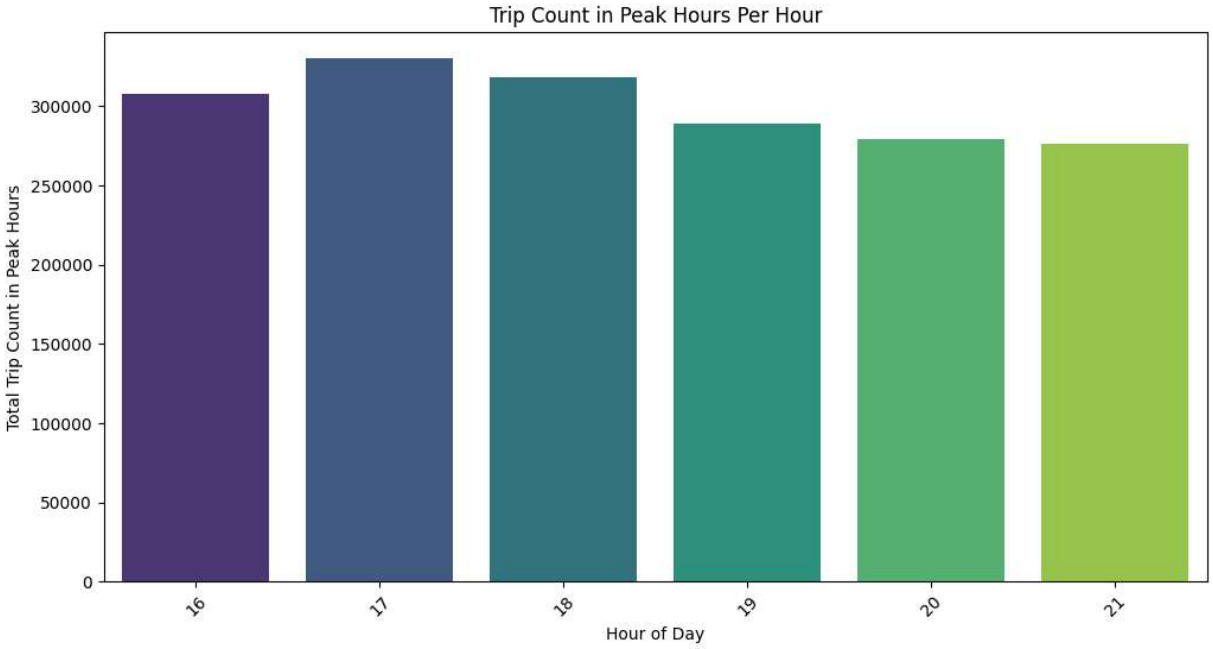


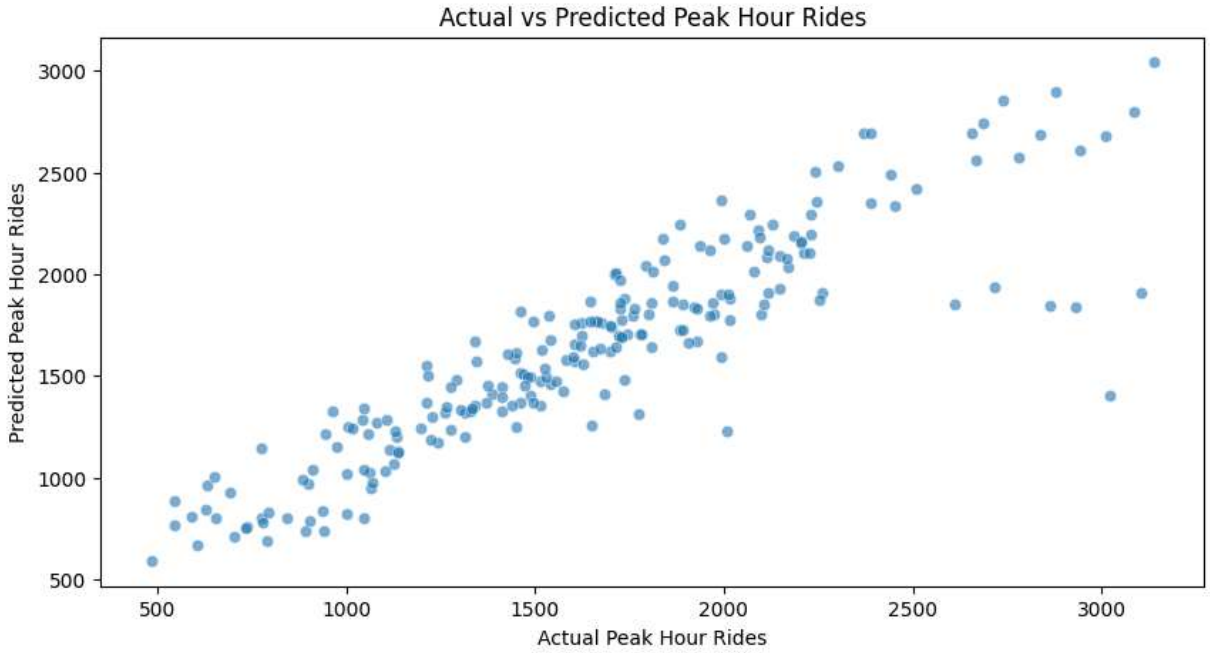




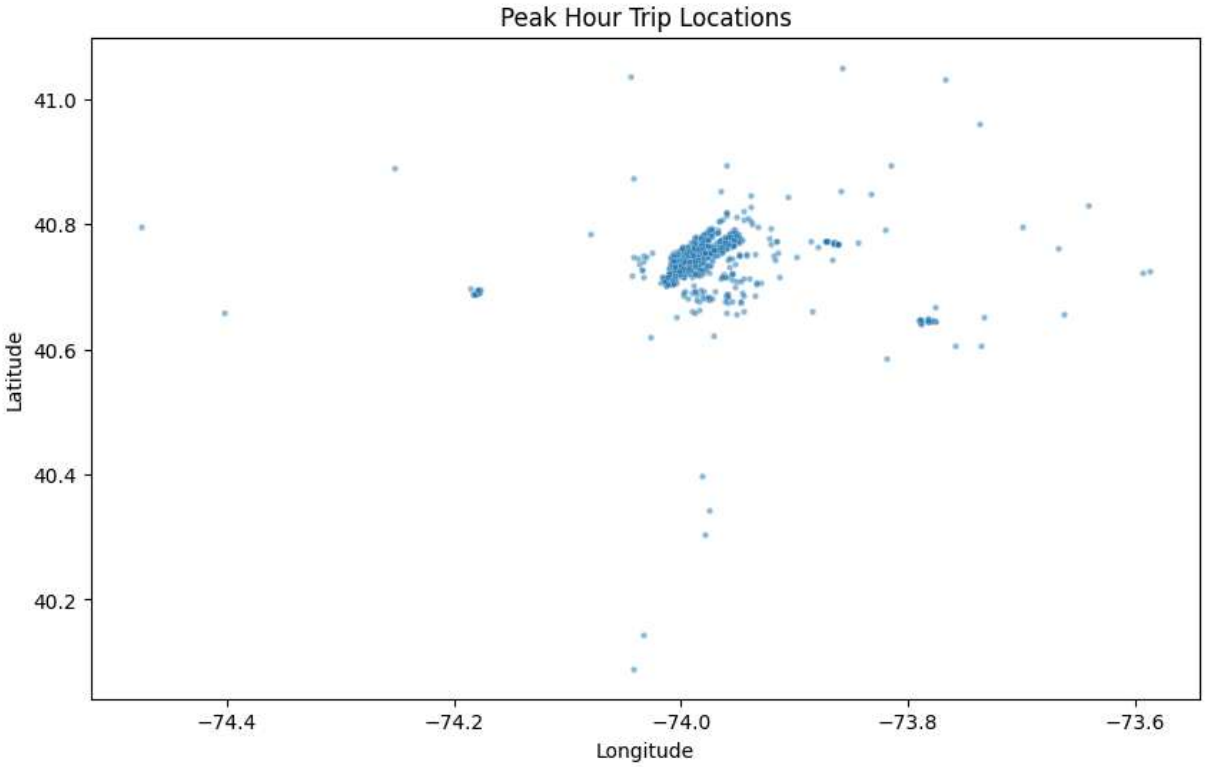


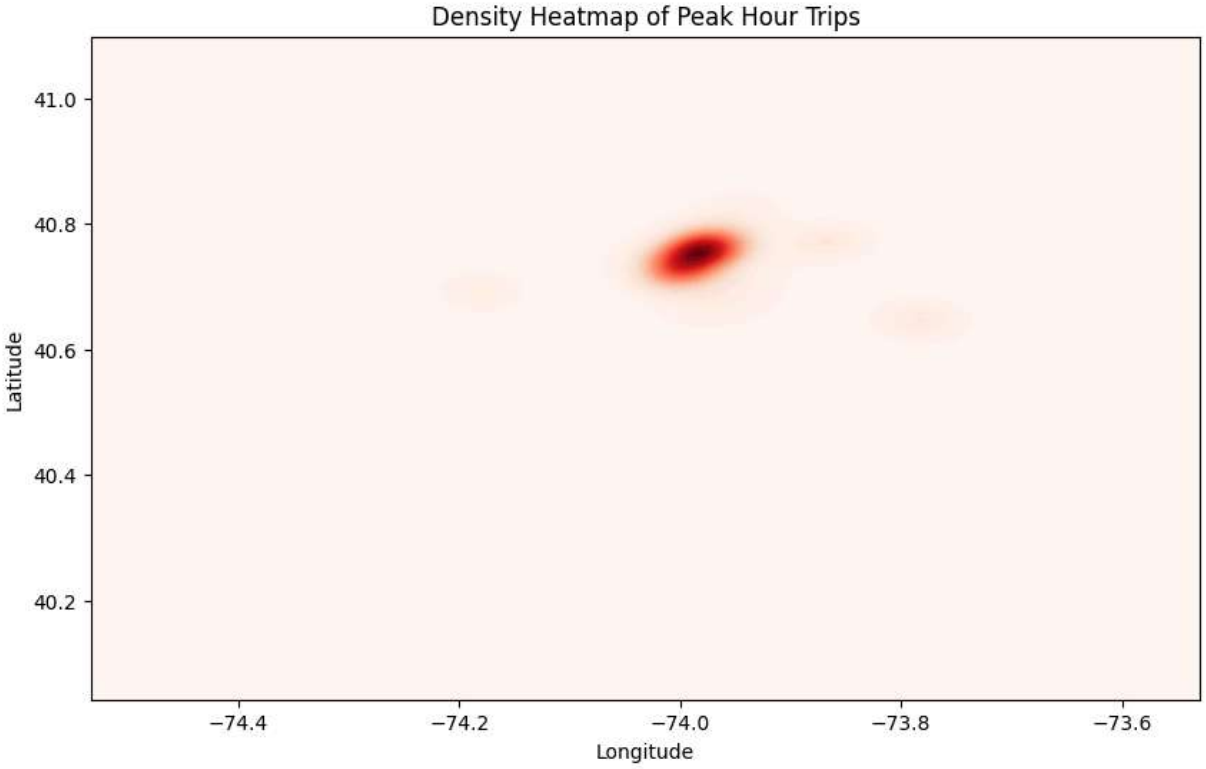

```
<ipython-input-22-72d9b1eec9ac>:126: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(data=peak_hour_hourly, x='Hour', y='Trip_Count', palette='viridis')
```



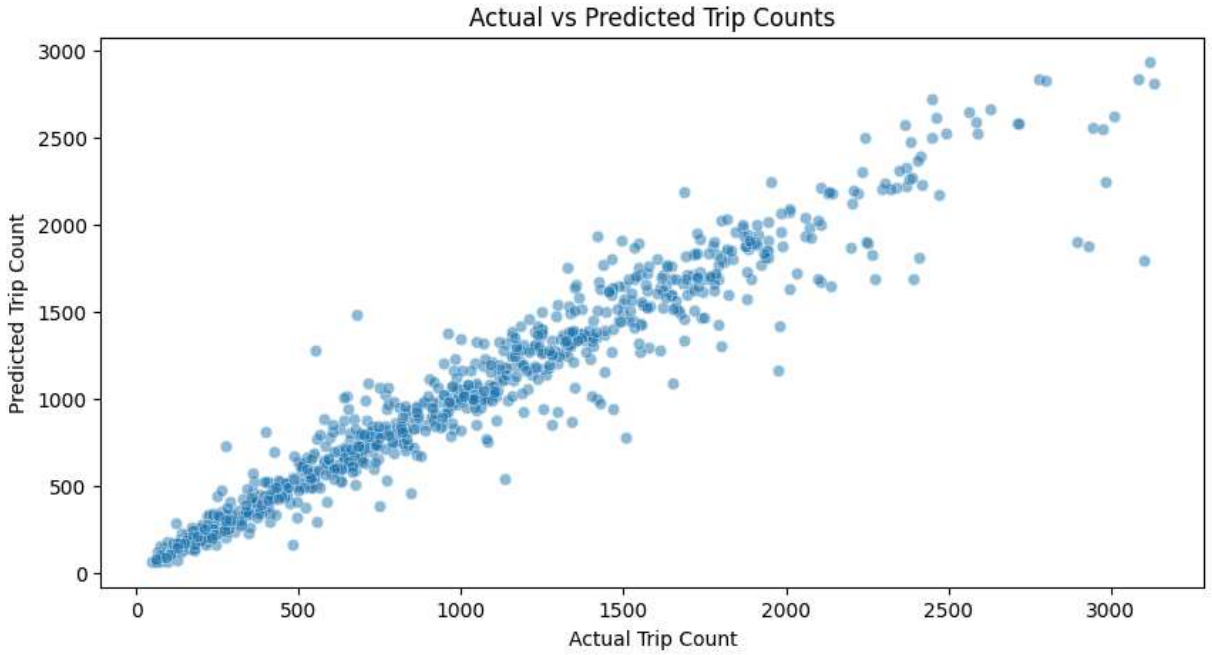


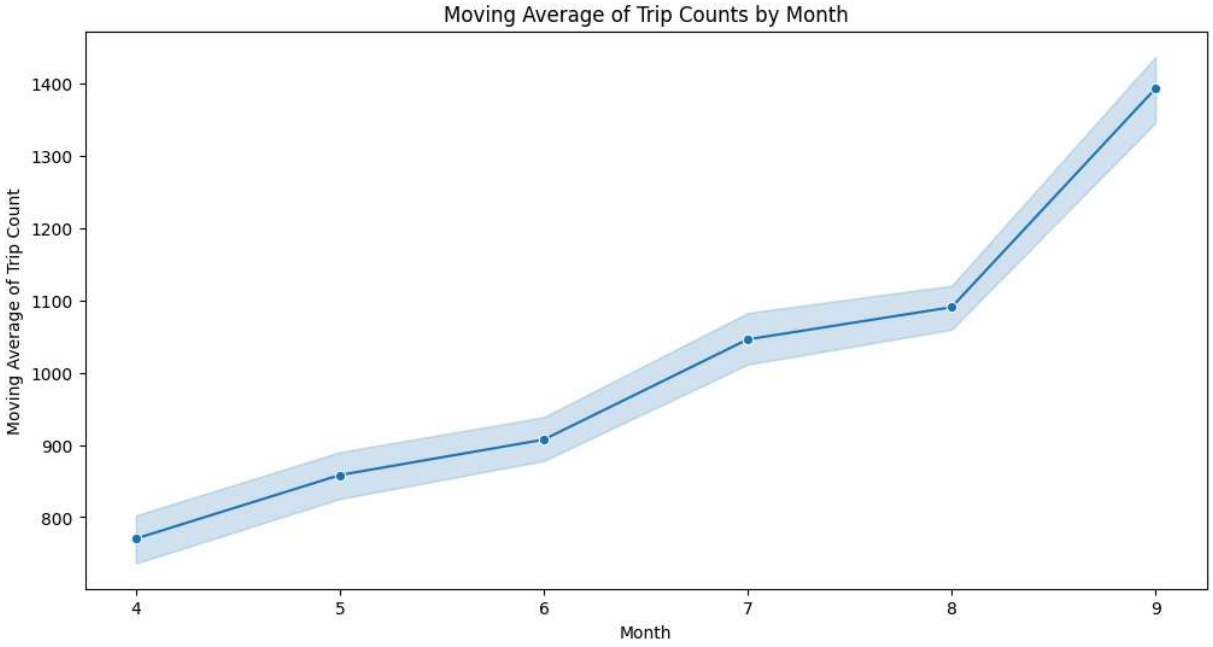

```
RMSE for peak hour rides -------------
RMSE for Peak Hour Rides Prediction: 254.70
Peak hour trips to Latitude , Longitude ---------------------
```






```
Trip analysis for all hours ---------------------------------
------------------------------------------------------------
RMSE for all hour rides :
RMSE: 166.30
```





---------- End of Random Forest Regressor Model ----------

---------- Long Short Term Memory (LSTM) Model ----------

In [23]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
```

```
# Load dataset
file_path = "/content/sample_data/uber-raw-data-aprsep-14-clean.csv"
df = pd.read_csv(file_path)

# Convert Date/Time to datetime format
df['Date/Time'] = pd.to_datetime(df['Date/Time'])

# Extract time-based features
df['Hour'] = df['Date/Time'].dt.hour
df['Day'] = df['Date/Time'].dt.day
df['Month'] = df['Date/Time'].dt.month
df['Weekday'] = df['Date/Time'].dt.weekday

# Aggregate data to create a time series of trip demand
df['Count'] = 1  # Each row is a trip, so count them
time_series = df.resample('h', on='Date/Time').sum()['Count'].fillna(0)  # Hourly a

# Normalize data
scaler = MinMaxScaler()
time_series_scaled = scaler.fit_transform(time_series.values.reshape(-1, 1))

# Prepare sequences for LSTM
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(X), np.array(y)

seq_length = 24  # Using past 24 hours to predict the next hour
X, y = create_sequences(time_series_scaled, seq_length)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=Fa

# Define LSTM Model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(seq_length, 1)),
    Dropout(0.2),
    LSTM(50, return_sequences=False),
    Dropout(0.2),
    Dense(25),
    Dense(1)
])


# Compile the model
model.compile(optimizer='adam', loss='mse')

# Print model summary
model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarnin
g: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequenti
al models, prefer using an `Input(shape)` object as the first layer in the model ins
tead.
  super().__init__(**kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	
lstm_2 (LSTM)	(None, 24, 50)	
dropout_2 (Dropout)	(None, 24, 50)	
lstm_3 (LSTM)	(None, 50)	
dropout_3 (Dropout)	(None, 50)	
dense_2 (Dense)	(None, 25)	
dense_3 (Dense)	(None, 1)	

Total params: 31,901 (124.61 KB)

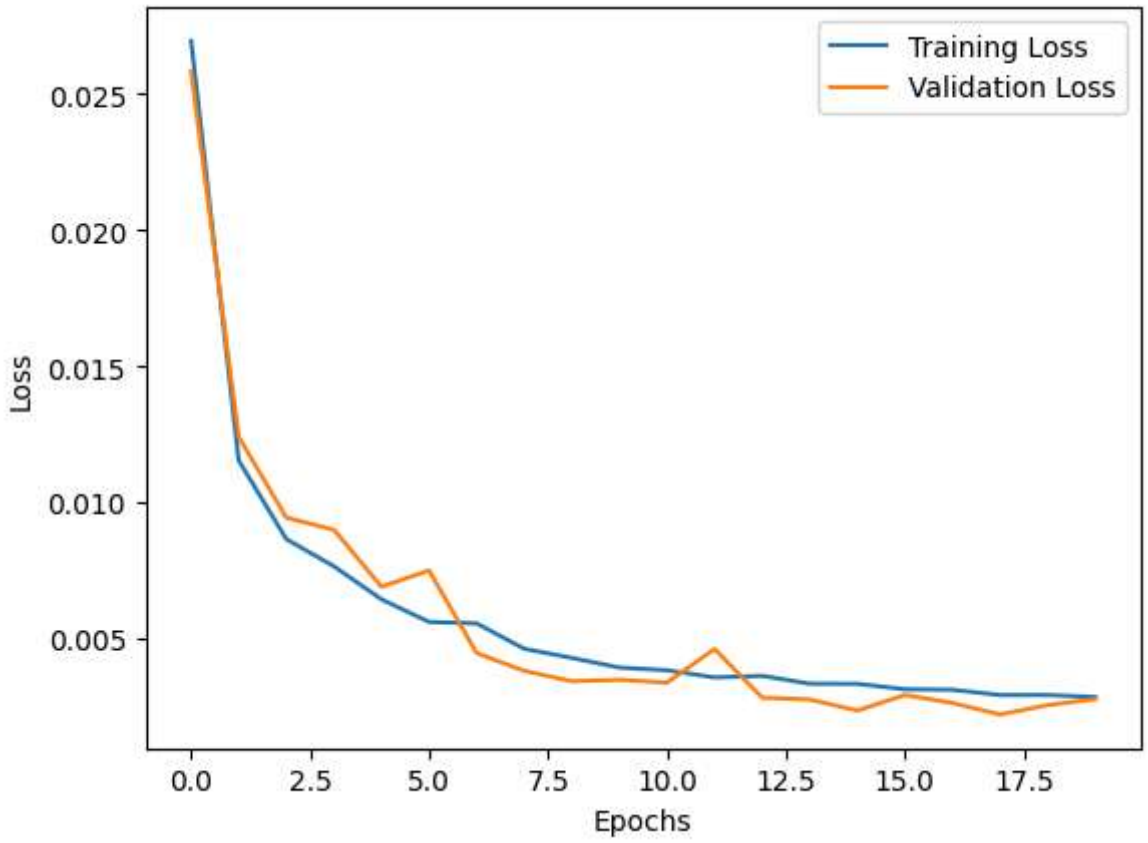
Trainable params: 31,901 (124.61 KB)

Non-trainable params: 0 (0.00 B)

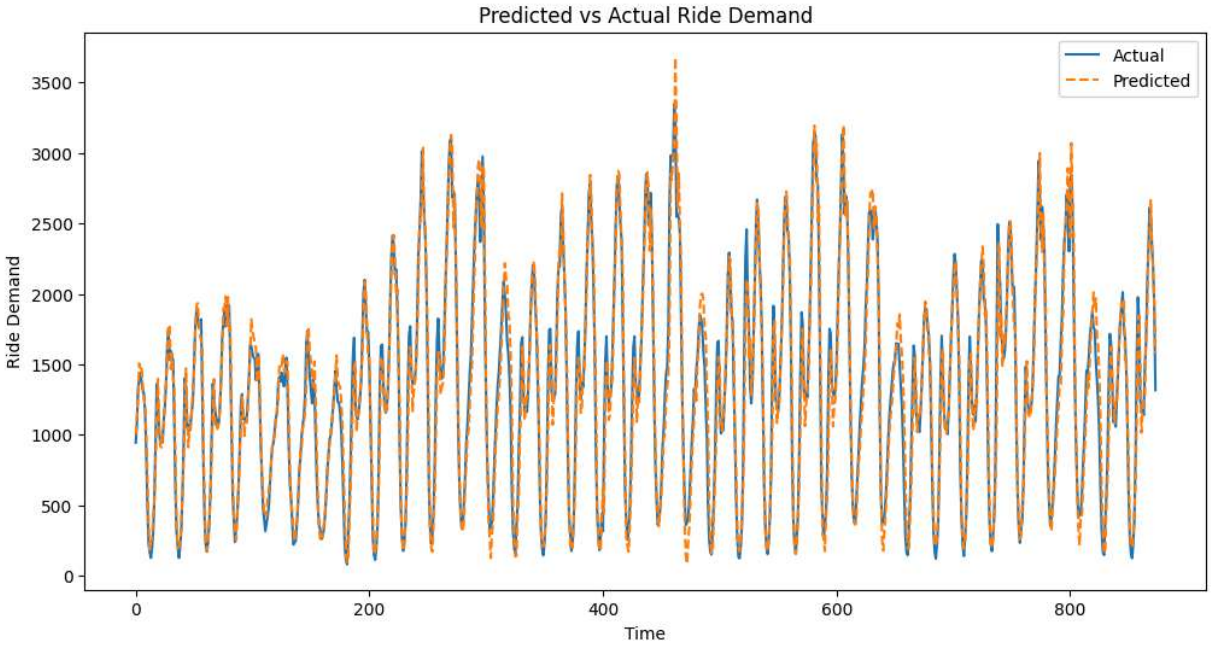
In [24]:
```
# Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_

# Plot training loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Predict and visualize results
y_pred = model.predict(X_test)
y_pred_rescaled = scaler.inverse_transform(y_pred)
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plot actual vs predicted values
plt.figure(figsize=(12, 6))
plt.plot(y_test_rescaled, label="Actual")
plt.plot(y_pred_rescaled, label="Predicted", linestyle='dashed')
plt.xlabel('Time')
plt.ylabel('Ride Demand')
plt.legend()
plt.title('Predicted vs Actual Ride Demand')
plt.show()

# Choose a smaller time window for better visualization
```

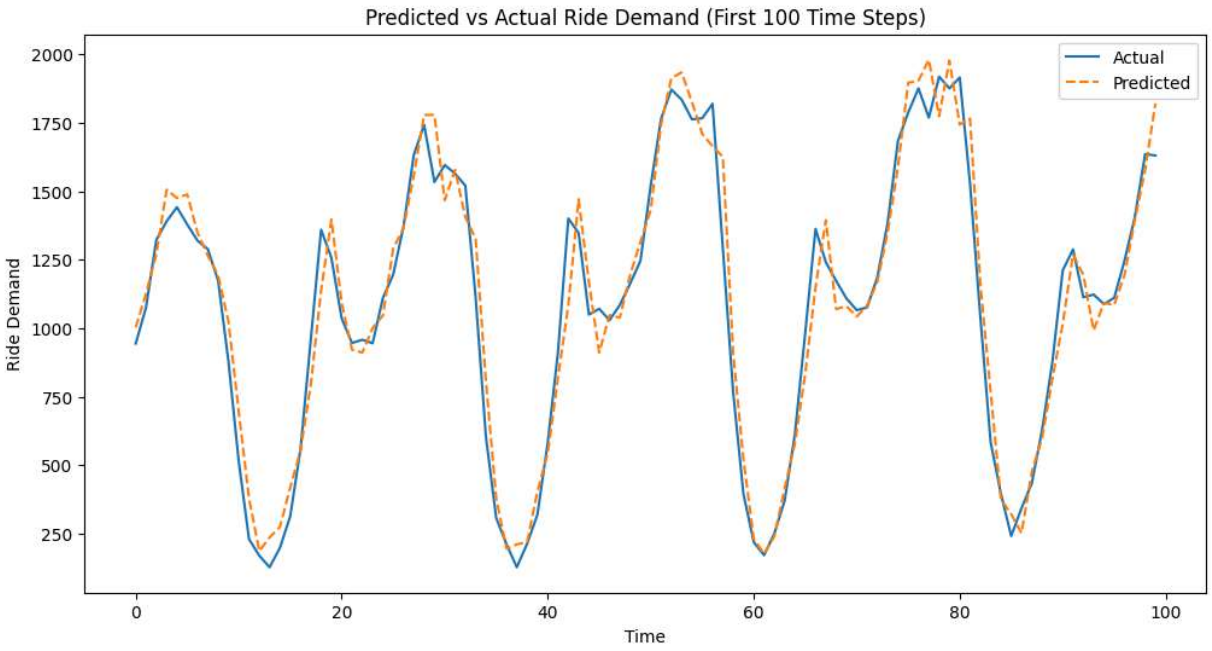
```
time_window = 100  # Adjust this value as needed

plt.figure(figsize=(12, 6))
plt.plot(y_test_rescaled[:time_window], label="Actual")
plt.plot(y_pred_rescaled[:time_window], label="Predicted", linestyle='dashed')
plt.xlabel('Time')
plt.ylabel('Ride Demand')
plt.legend()
plt.title(f'Predicted vs Actual Ride Demand (First {time_window} Time Steps)')
plt.show()
```

```
Epoch 1/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 9s 42ms/step - loss: 0.0395 - val_loss: 0.0258
Epoch 2/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 34ms/step - loss: 0.0127 - val_loss: 0.0124
Epoch 3/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 28ms/step - loss: 0.0089 - val_loss: 0.0094
Epoch 4/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 3s 28ms/step - loss: 0.0079 - val_loss: 0.0090
Epoch 5/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 6s 33ms/step - loss: 0.0066 - val_loss: 0.0069
Epoch 6/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 5s 30ms/step - loss: 0.0057 - val_loss: 0.0075
Epoch 7/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 6s 39ms/step - loss: 0.0061 - val_loss: 0.0045
Epoch 8/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 3s 30ms/step - loss: 0.0049 - val_loss: 0.0038
Epoch 9/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 5s 28ms/step - loss: 0.0045 - val_loss: 0.0034
Epoch 10/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 6s 40ms/step - loss: 0.0041 - val_loss: 0.0035
Epoch 11/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 26ms/step - loss: 0.0037 - val_loss: 0.0034
Epoch 12/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 3s 26ms/step - loss: 0.0035 - val_loss: 0.0046
Epoch 13/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 3s 27ms/step - loss: 0.0041 - val_loss: 0.0028
Epoch 14/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 38ms/step - loss: 0.0033 - val_loss: 0.0028
Epoch 15/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 36ms/step - loss: 0.0033 - val_loss: 0.0024
Epoch 16/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 28ms/step - loss: 0.0032 - val_loss: 0.0029
Epoch 17/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 6s 40ms/step - loss: 0.0033 - val_loss: 0.0026
Epoch 18/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 4s 31ms/step - loss: 0.0031 - val_loss: 0.0022
Epoch 19/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 3s 26ms/step - loss: 0.0030 - val_loss: 0.0026
Epoch 20/20
110/110 ━━━━━━━━━━━━━━━━━━━━ 6s 33ms/step - loss: 0.0027 - val_loss: 0.0028
```




```
28/28 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step
```





In [25]:
```
#y_pred = model.predict(X_test)  # Predictions from LSTM
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error of LSTM Model: {mse}")
```

```
Mean Squared Error of LSTM Model: 0.0027730060409103465
```

------- End of Long Short Term Memory Model ----------

---------- End of Code ----------

Result:

Identified Peak Hours: [16, 17, 18, 19, 20, 21]

1. Long Short Term Memory (LSTM) Model:

   (a) Mean Squared Error: 0.0023550111335187657

2. Random Forest Regressor:

   (a) RMSE for all hour rides : 166.30

   (b) RMSE for Peak Hour Rides Prediction: 254.70

---------- End of document ----------