DASC 5300_Foc_Assignment_2

Data Pre-processing:

For the first part of the assignment, you will be working with a subset of the dataset containing 100,000 randomly sampled rows. Use the last 4 digits of your student ID as the random seed to ensure consistency across submissions. You can use the following code to load the dataset and perform the sampling:

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from geopy.distance import geodesic
import seaborn as sns
import matplotlib.pyplot as plt
import datetime as dt
```

import time

from IPython.display import clear_output

pandas (pd): Used for data manipulation and analysis.

numpy (np): Provides support for large, multi-dimensional arrays and matrices, along with mathematical functions.

scikit-learn (sklearn): A machine learning library in Python. You've imported various regression models and metrics for model evaluation.

XGBoost: An optimized distributed gradient boosting library.

geopy: Used for geocoding and calculating distances between geographical coordinates.

seaborn: A data visualization library based on Matplotlib, used for statistical graphics.

matplotlib.pyplot (plt): A plotting library for creating static, animated, and interactive visualizations in Python.

datetime: Provides classes for working with dates and times.

time: Provides various time-related functions.

IPython.display: Provides utilities for displaying interactive widgets in the IPython notebook.

Load the dataset

df = pd.read_csv('train.csv')

Set the random seed

np.random.seed(8659)

Sample 100,000 rows

df = df.sample(n=100000)

1.Load Dataset: Read a CSV file named 'train.csv' into a pandas DataFrame using pd.read_csv('train.csv').

2.Set Random Seed: Set the random seed to ensure reproducibility in data sampling with np.random.seed(8659).

3.Sample Rows: Randomly sample 100,000 rows from the DataFrame using df.sample(n=100000) and overwrite the original DataFrame with the sampled data.

df.head()

Display the first few rows of the DataFrame 'df' using the head() function to get a quick overview of the data structure and content in Python pandas.

Output :- id vendor_id pickup_datetime dropoff_datetime passenger_count pickup_longitude pickup_latitude dropoff_longitude dropoff_latitude store_and_fwd_flag trip_duration

```
209174      id2104500 2      2016-03-09 20:09:32  2016-03-09   20:33:18
     5      -74.00    40.74       -73.96      40.67        N      1426

1296156     id1063788 2      2016-05-26 08:38:50  2016-05-26   08:51:19
     1      -73.97    40.74       -73.99      40.75        N      749

136770      id1376921 1      2016-06-12 10:36:15  2016-06-12   10:50:21
     1      -73.99    40.73       -74.00      40.71        N      846

219673      id1418169 2      2016-06-26 02:13:21  2016-06-26   02:16:28
     1      -73.98    40.74       -73.98      40.73        N      187

218117      id2470055 2      2016-02-17 11:33:38  2016-02-17   11:39:45
     1      -73.98    40.73       -73.99      40.73        N      367
```

df.describe()

Generate descriptive statistics of the DataFrame 'df' using the describe() function in Python pandas. This provides statistical summary including count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum values for each numeric column.

Output:-

```
    index  vendor_id  passenger_count      pickup_longitude
     pickup_latitude  dropoff_longitude    dropoff_latitude
     trip_duration

count       100000.0   100000.0   100000.0   100000.0   100000.0
     100000.0   100000.0
```

mean 1.53466 1.66315 -73.97355921989441
40.75100309143066 -73.97339245399475 40.751922385520935
942.74166

std 0.49879973176789516 1.3120744235343245
0.04205273835032488 0.03688510034430534
0.04430950564256595 0.04172317629762456 5
3083.431292395826

min 1.0 0.0 -77.4407501220703 35.310306549072266 -
79.51861572265625 35.173545837402344 1.0

25% 1.0 1.0 -73.99192047119139 40.73754119873047 -
73.99134063720702 40.73601913452149 394.0

50% 2.0 1.0 -73.98172760009764 40.75435256958008 -
73.9797134399414 40.75481033325195 661.0

75% 2.0 2.0 -73.96732330322266 40.76845932006836 -
73.96297454833984 40.769981384277344 1073.0

max 2.0 6.0 -70.51190185546875 42.45894241333008 -
70.51190185546875 43.92102813720703 86369.0

df.dtypes

Inspect the data types of each column in the DataFrame 'df' using **df.dtypes** to get a summary of the variable types (e.g., int, float, object) present in the dataset.

Output:- id object

```
vendor_id             int64
pickup_datetime       object
dropoff_datetime      object
passenger_count        int64
pickup_longitude     float64
pickup_latitude      float64
dropoff_longitude    float64
dropoff_latitude     float64
store_and_fwd_flag    object
trip_duration          int64
dtype: object
```

```
## Droping rows with nan values
df.dropna(inplace=True)
```

Remove rows with missing values (NaN) from the DataFrame 'df' in-place using the dropna() function with inplace=True. This modifies the DataFrame directly, eliminating any rows containing null values.

```
null_data = df.isnull().sum()
print("null data counts:")
print(null_data)
```

Calculate and print the count of null values for each column in the DataFrame 'df' using isnull().sum(). The resulting 'null_data' series provides a summary of the number of missing values in each column.

Output:-

```
null data counts:
id                  0
vendor_id           0
pickup_datetime     0
dropoff_datetime    0
passenger_count     0
pickup_longitude    0
pickup_latitude     0
dropoff_longitude   0
dropoff_latitude    0
store_and_fwd_flag  0
trip_duration       0
dtype: int64
```

df.head()

```
id        vendor_id  pickup_datetime        dropoff_datetime
    passenger_count        pickup_longitude        pickup_latitude
    dropoff_longitude      dropoff_latitude store_and_fwd_flag
```

```
        trip_duration   month   weekday   weekday_num
        pickup_hour

209174    id2104500 2    2016-03-09 20:09:32  2016-03-09 20:33:18
        5    -74.000259    40.737808 -73.964043    40.674789 N
        1426 3    Wednesday    2    20

1296156   id1063788 2    2016-05-26 08:38:50  2016-05-26 08:51:19
        1    -73.974258    40.742870 -73.988464    40.747341 N
        749  5    Thursday  3    8

136770    id1376921 1    2016-06-12 10:36:15  2016-06-12 10:50:21
        1    -73.994003    40.734497 -73.998840    40.714928 N
        846  6    Sunday    6    10

219673    id1418169 2    2016-06-26 02:13:21  2016-06-26 02:16:28
        1    -73.979652    40.739578 -73.975937    40.732571 N
        187  6    Sunday    6    2

218117    id2470055 2    2016-02-17 11:33:38  2016-02-17 11:39:45
        1    -73.982903    40.726830 -73.990150    40.734669 N
        367  2    Wednesday    2    11
```

```
df = df[df['trip_duration'] >= 0]
```

Filter the DataFrame 'df' to include only rows where the 'trip_duration' column has a value greater than or equal to 0. This operation retains only the rows where the trip duration is non-negative.

```
df['pickup_datetime']=pd.to_datetime(df['pickup_datetime'])
```

```
df['dropoff_datetime'] = pd.to_datetime(df['dropoff_datetime'])
```

Convert the 'pickup_datetime' and 'dropoff_datetime' columns in the DataFrame 'df' to datetime format using pd.to_datetime(). This ensures that the specified columns are interpreted as datetime objects, allowing for convenient manipulation and analysis of temporal data.

```
df['month'] = df.pickup_datetime.dt.month
df['weekday'] = df['pickup_datetime'].dt.strftime('%A')
df['weekday_num'] = df.pickup_datetime.dt.weekday
df['pickup_hour'] = df.pickup_datetime.dt.hour
```

Extract Month: Create a new 'month' column in the DataFrame 'df' by extracting the month from the 'pickup_datetime' column using df.pickup_datetime.dt.month.

Extract Weekday (Full Name): Add a 'weekday' column to 'df' by converting the 'pickup_datetime' column to a full weekday name (e.g., Monday, Tuesday) using df['pickup_datetime'].dt.strftime('%A').

Extract Weekday (Numeric): Introduce a 'weekday_num' column by extracting the numeric representation of the weekday (0 for Monday, 1 for Tuesday, and so on) from the 'pickup_datetime' column using df.pickup_datetime.dt.weekday.

Extract Pickup Hour: Generate a 'pickup_hour' column by extracting the hour component from the 'pickup_datetime' column using df.pickup_datetime.dt.hour.

df.head()

id vendor_id pickup_datetime dropoff_datetime passenger_count pickup_longitude pickup_latitude dropoff_longitude dropoff_latitude store_and_fwd_flag trip_duration month weekday weekday_num pickup_hour

209174 id2104500 2 2016-03-09 20:09:32 2016-03-09 20:33:18 5 -74.00 40.74 -73.96 40.67 N 1426 3 Wednesday 2 20

1296156 id1063788 2 2016-05-26 08:38:50 2016-05-26 08:51:19 1 -73.97 40.74 -73.99 40.75 N 749 5 Thursday 3 8

136770 id1376921 1 2016-06-12 10:36:15 2016-06-12 10:50:21 1 -73.99 40.73 -74.00 40.71 N 846 6 Sunday 6 10

219673 id1418169 2 2016-06-26 02:13:21 2016-06-26 02:16:28 1 -73.98 40.74 -73.98 40.73 N 187 6 Sunday 6 2

218117 id2470055 2 2016-02-17 11:33:38 2016-02-17 11:39:45 1 -73.98 40.73 -73.99 40.73 N 367 2 Wednesday 2 11

df.dtypes

Check and display the data types of each column in the DataFrame 'df' using df.dtypes. This provides an overview of the variable types present in the dataset, helping you understand how the data is stored and interpreted.

Output:-
```
id                    object
vendor_id             int64
pickup_datetime       datetime64[ns]
dropoff_datetime      datetime64[ns]
passenger_count       int64
pickup_longitude      float64
pickup_latitude       float64
dropoff_longitude     float64
dropoff_latitude      float64
store_and_fwd_flag    object
trip_duration         int64
month                 int64
weekday               object
weekday_num           int64
pickup_hour           int64
dtype: object
```

df.value_counts()

Use df.value_counts() to obtain the count of unique values in each column of the DataFrame 'df'. This function is particularly useful for categorical variables, providing a quick summary of the distribution of values within each column. Note that it is applied to each series individually.

id vendor_id pickup_datetime dropoff_datetime passenger_count pickup_longitude pickup_latitude dropoff_longitude dropoff_latitude store_and_fwd_flag trip_duration month weekday weekday_num pickup_hour

id0000063 2 2016-04-24 10:38:59 2016-04-24 10:46:34 1 -73.96 40.78 -73.98 40.77 N 455 4 Sunday 6 10 1

id2660312 2 2016-06-24 17:38:43 2016-06-24 18:07:03 1 -73.95 40.77 -73.99 40.74 N 1700 6 Friday 4 17 1

id2661341 1 2016-05-19 19:37:45 2016-05-19 19:59:57 1 -74.01 40.72 -74.01 40.72 N 1332 5 Thursday 3 19 1

id2661334 1 2016-02-14 03:54:20 2016-02-14 04:09:01 1 -73.99 40.76 -73.93 40.76 N 881 2 Sunday 6 3 1

id2661279 1 2016-03-03 22:22:14 2016-03-03 22:49:38 1 -73.87 40.77 -74.00 40.72 N 1644 3 Thursday 3 22 1

..

id1333253 1 2016-02-13 11:50:07 2016-02-13 12:06:04 1
-74.00 40.76 -73.97 40.75 Y 957
2 Saturday 5 11 1

id1333117 1 2016-04-26 21:06:19 2016-04-26 21:30:12 1
-73.87 40.77 -74.01 40.72 N 1433
4 Tuesday 1 21 1

id1333107 1 2016-02-26 23:00:23 2016-02-26 23:20:27 1
-73.98 40.77 -74.02 40.70 N 1204
2 Friday 4 23 1

id1333076 2 2016-01-30 17:02:44 2016-01-30 17:17:05 1
-73.97 40.76 -73.98 40.78 N 861
1 Saturday 5 17 1

id3999962 2 2016-05-05 12:11:15 2016-05-05 12:14:19 1
-73.96 40.77 -73.95 40.78 N 184
5 Thursday 3 12 1

Name: count, Length: 100000, dtype: int64

Creating dummy variables for store_and_fwd_flag within df2 and droping the first level

```
df = pd.get_dummies(df, columns=['store_and_fwd_flag'], drop_first=True)
```

Apply one-hot encoding to the 'store_and_fwd_flag' column in the DataFrame 'df' using pd.get_dummies(). This creates binary (0 or 1) indicator columns for each category in 'store_and_fwd_flag', and the original column is dropped (drop_first=True) to avoid multicollinearity in certain models. This transformation is useful when dealing with categorical data in machine learning.

```
# Create a figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Plot a bar chart in the first subplot
sns.countplot(data=df, x='vendor_id', ax=axes[0], palette='viridis')
axes[0].set_title('Vendors (Bar Chart)')
axes[0].set_xlabel('Vendor Id')
axes[0].set_ylabel('Count')

# Plot a pie chart in the second subplot
vendor_counts = df['vendor_id'].value_counts()
axes[1].pie(vendor_counts, labels=vendor_counts.index, autopct='%1.1f%%', startangle=90, colors=sns.color_palette('viridis'))
axes[1].set_title('Vendors (Pie Chart)')
```

```
# Adjust layout
fig.tight_layout()

# Show the plots
plt.show()
```

In this Python code snippet using Matplotlib and Seaborn, a figure with two subplots is created to visualize the distribution of 'vendor_id' in a dataset:

Figure and Subplots Creation:

A figure with two subplots is created using plt.subplots(1, 2, figsize=(12, 5)).

Bar Chart (First Subplot):

A bar chart is plotted in the first subplot using Seaborn's countplot().

The 'vendor_id' column is used for the x-axis, and the 'viridis' color palette is applied.

Subplot title, x-axis label, and y-axis label are set using set_title(), set_xlabel(), and set_ylabel().

Pie Chart (Second Subplot):

A pie chart is plotted in the second subplot.

Vendor counts are calculated using value_counts() and stored in the 'vendor_counts' variable.

The pie chart is created using plt.pie() with labels, autopct for percentage display, startangle for rotation, and colors from the 'viridis' palette.

Subplot title is set with set_title().

Layout Adjustment:

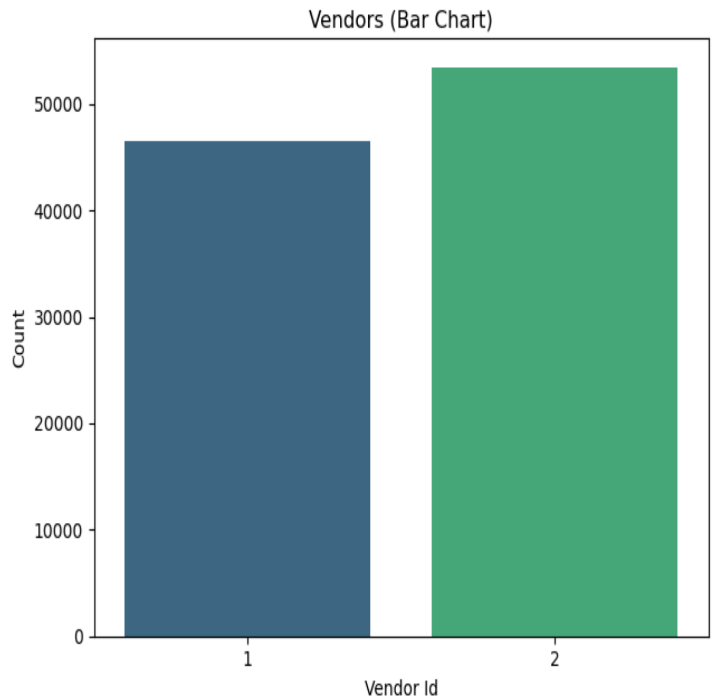
Adjust the layout for better visualization using fig.tight_layout().

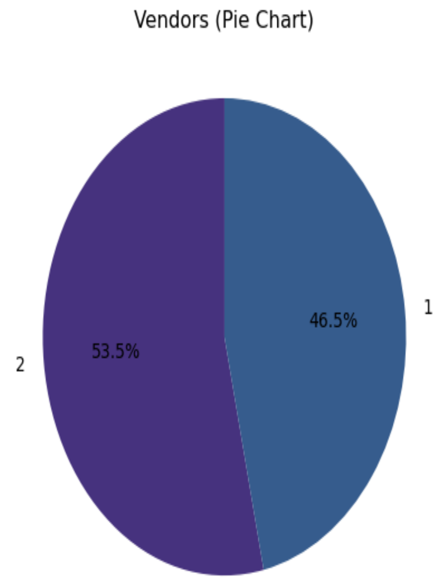
Display Plots:

Finally, display the plots using plt.show().

This code provides a visual representation of vendor distribution with a bar chart and a pie chart, aiding in data exploration and understanding.

Output:-






```
# Converting 'passenger_count' to integers and then count the values
df['passenger_count'] = df['passenger_count'].astype(int)
passenger_count_counts = df['passenger_count'].value_counts()

# Seting the display format to suppress scientific notation
pd.options.display.float_format = '{:.2f}'.format

# Display the value counts
```

```
print(passenger_count_counts)
```

- Convert Passenger Count to Integers:

The 'passenger_count' column in the DataFrame 'df' is converted to integers using astype(int).

Count Passenger Count Values:

Calculate the count of each unique passenger count using value_counts() and store the result in the 'passenger_count_counts' variable.

Set Display Format:

Set the display format to suppress scientific notation for better readability using pd.options.display.float_format = '{:.2f}'.format.

Display Value Counts:

Display the counts of passenger counts with print(passenger_count_counts).

This code snippet ensures 'passenger_count' is treated as integers, calculates and prints the count of each unique passenger count, and adjusts the display format for better readability.

```python
# Converting 'passenger_count' to integers and then count the values
df['passenger_count'] = df['passenger_count'].astype(int)
passenger_count_counts = df['passenger_count'].value_counts()

# Seting the display format to suppress scientific notation
pd.options.display.float_format = '{:.2f}'.format

# Display the value counts
print(passenger_count_counts)
```

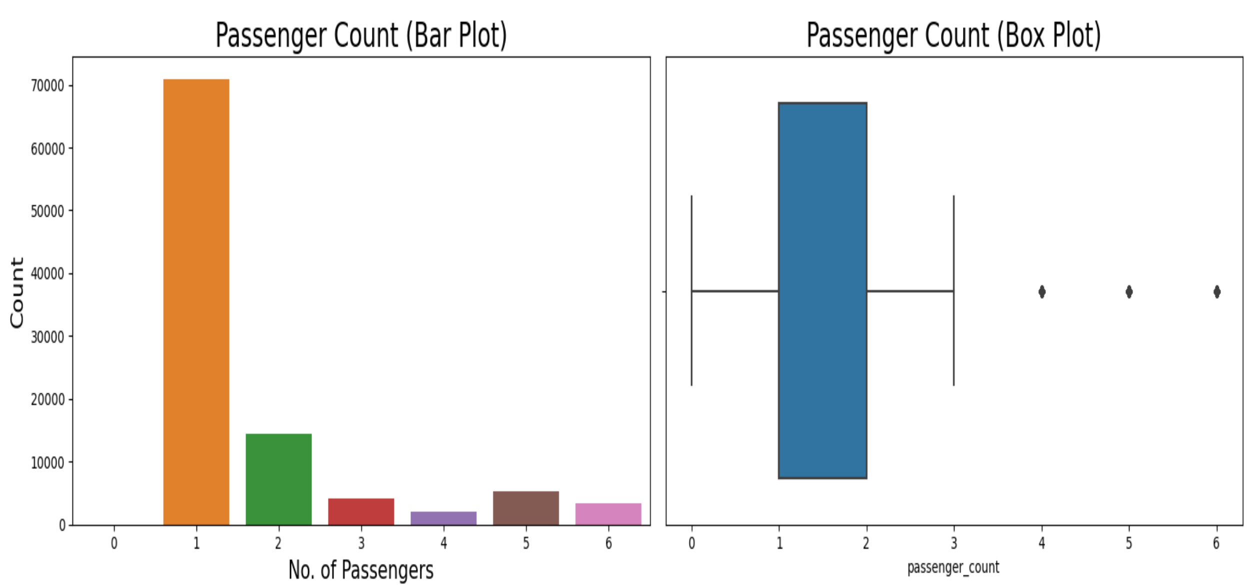
```
passenger_count
1    70883
2    14403
5     5316
3     4096
6     3289
4     2006
0        7
Name: count, dtype: int64
```

df = df[df['passenger_count'] > 0]

Filter the DataFrame 'df' to include only rows where the 'passenger_count' column has values greater than 0. This operation removes rows with non-positive passenger counts, ensuring the dataset only contains valid entries.

```python
# Create a single figure with two subplots side by side
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 5))

# Bar plot for passenger count
sns.countplot(data=df, x='passenger_count', ax=axes[0])
axes[0].set_ylabel("Count", fontsize=15)
axes[0].set_xlabel("No. of Passengers", fontsize=15)
axes[0].set_title('Passenger Count (Bar Plot)', fontsize=20)

# Box plot for passenger count
sns.boxplot(data=df, x='passenger_count', orient='h', ax=axes[1])
axes[1].set_title('Passenger Count (Box Plot)', fontsize=20)

# Adjust layout
fig.tight_layout()

# Show the plots
plt.show()
```

Output:-



Create Figure and Subplots:

A single figure with two subplots arranged side by side is created using plt.subplots(nrows=1, ncols=2, figsize=(16, 5)).

Bar Plot (First Subplot):

A bar plot for the 'passenger_count' column is generated in the first subplot using Seaborn's countplot().

Y-axis represents the count, and labels, title, and font sizes are set for clarity.

Box Plot (Second Subplot):

A box plot for 'passenger_count' is created in the second subplot using Seaborn's boxplot().

The orientation is set to horizontal ('h') for better visualization.

Subplot title and font size are adjusted.

Layout Adjustment:

The layout is adjusted for a cleaner appearance with fig.tight_layout().

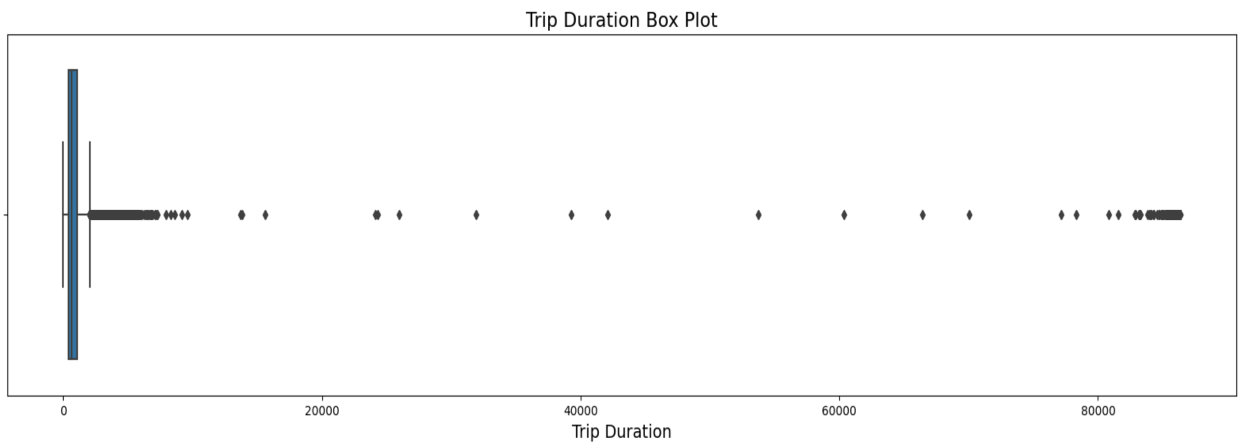
Display Plots:

Finally, both plots are displayed using plt.show().

This code provides a visual exploration of the distribution of passenger counts through a bar plot and a box plot, offering insights into the variability and central tendency of the data.

```python
# Replace 'df' with your actual DataFrame
plt.figure(figsize=(20, 5))
sns.boxplot(x=df['trip_duration'])
plt.title('Trip Duration Box Plot', fontsize=16)
plt.xlabel('Trip Duration', fontsize=14)
plt.show()
```

Output:-



Create Figure and Set Size:

A figure with a specific size is created using plt.figure(figsize=(20, 5)).

Box Plot for 'trip_duration':

A box plot for the 'trip_duration' column is generated using Seaborn's boxplot().

Title and Axis Labels:

The plot title, x-axis label, and font sizes are set for better readability.

Display the Plot:

Finally, the box plot is displayed using plt.show().

This code snippet visualizes the distribution of 'trip_duration' through a box plot, allowing for an exploration of the central tendency, variability, and potential outliers in the dataset. Adjusting the figure size and adding clear labels enhances the interpretability of the plot.

```
bin_edges = np.arange(0, df['trip_duration'].max(), 3600)

# Group and count trips based on trip duration bins
trip_counts = df['trip_duration'].groupby(pd.cut(df['trip_duration'], bin_edges)).count()

# Print the trip counts
print(trip_counts)
```

```
trip_duration
(0, 3600]         99176
(3600, 7200]        680
(7200, 10800]         7
(10800, 14400]        2
(14400, 18000]        1
(18000, 21600]        0
(21600, 25200]        2
(25200, 28800]        1
(28800, 32400]        1
(32400, 36000]        0
(36000, 39600]        1
(39600, 43200]        1
(43200, 46800]        0
(46800, 50400]        0
(50400, 54000]        1
(54000, 57600]        0
(57600, 61200]        1
(61200, 64800]        0
(64800, 68400]        1
(68400, 72000]        1
(72000, 75600]        0
(75600, 79200]        2
(79200, 82800]        2
Name: trip_duration, dtype: int64
```

In this Python code snippet using NumPy and pandas:

Define Bin Edges:

Bin edges for grouping trip durations are defined using np.arange(0, df['trip_duration'].max(), 3600). Here, bins represent one-hour intervals.

Group and Count Trips:

The 'trip_duration' column is grouped into bins using pd.cut() based on the defined bin edges. The counts of trips in each bin are calculated using groupby() and count().

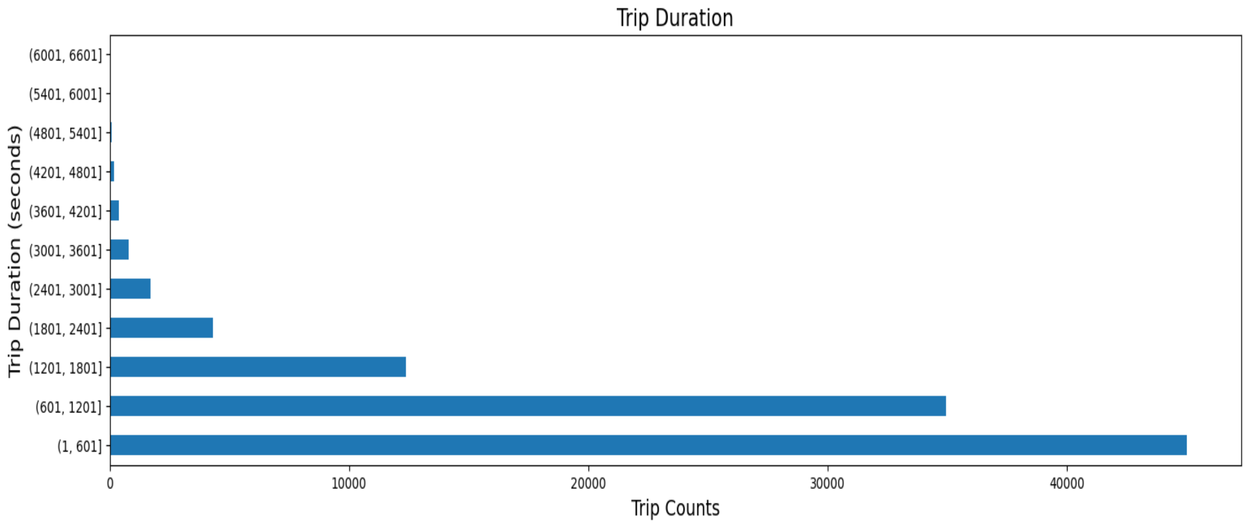
Print Trip Counts:

The resulting trip counts are printed using print(trip_counts)

This code snippet demonstrates the creation of bins for trip duration, grouping trips accordingly, and printing the counts for each bin, enabling an analysis of the distribution of trip durations in the dataset.

```
# Replace 'df' with your actual DataFrame
bin_labels = np.arange(1, 7200, 600)
trip_duration_counts = df['trip_duration'].groupby(pd.cut(df['trip_duration'], bin_labels)).count()

# Create a horizontal bar plot
plt.figure(figsize=(18, 5))
trip_duration_counts.plot(kind='barh')
plt.title('Trip Duration', fontsize=16)
plt.xlabel('Trip Counts', fontsize=14)
plt.ylabel('Trip Duration (seconds)', fontsize=14)
plt.show()
```

Output:-



In this Python code snippet using NumPy, pandas, and Matplotlib:

Define Bin Labels:

Bin labels for grouping trip durations are defined using np.arange(1, 7200, 600). Here, bins represent 10-minute intervals.

Group and Count Trips:

The 'trip_duration' column is grouped into bins using pd.cut() based on the defined bin labels. The counts of trips in each bin are calculated using groupby() and count().

Create Horizontal Bar Plot:

A horizontal bar plot is created using Matplotlib with trip_duration_counts.plot(kind='barh').

Plot title, x-axis label, y-axis label, and font sizes are set for better visualization.

Display the Plot:

Finally, the horizontal bar plot is displayed using plt.show()

```
def clock(ax, radii, title, color):
    N = 24  # Number of hours in a day
    bottom = 2  # Bottom position for the bars

    # Create theta for 24 hours
    theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)

    # Width of each bin on the plot
    width = (2 * np.pi) / N

    # Create bars on the polar plot
    bars = ax.bar(theta, radii, width=width, bottom=bottom, color=color, edgecolor="#999999")

    # Set the label position to start from the top and go clockwise
    ax.set_theta_zero_location("N")  # "N" stands for North (top)
    ax.set_theta_direction(-1)  # Clockwise direction

    # Set the label ticks and format them as hours
    ax.set_xticks(theta)
    ticks = ["{}:00".format(x) for x in range(24)]
    ax.set_xticklabels(ticks)

    # Set the title of the polar plot
    ax.set_title(title)
```

The provided Python function, clock(ax, radii, title, color), creates a polar plot (clock-like visualization) using Matplotlib. Here's a short note:

The clock function takes four parameters:

ax: The axes on which the polar plot will be created.

radii: An array of values representing the lengths of the bars in each hour bin.

title: The title of the polar plot.

color: The color of the bars in the plot.

The function generates a polar bar plot with 24 bars, each representing an hour of the day. It sets the zero location at the top of the plot (North) and labels the hours clockwise. The bars' lengths are determined by the radii parameter. The resulting plot provides a visual representation of values across 24 hours, making it suitable for displaying time-related data.

```
plt.figure(figsize=(15, 15))
ax = plt.subplot(3, 3, 1, polar=True)

# Calculate the number of trips per hour and convert it to an array
radii = df['pickup_hour'].value_counts().sort_index().values

title = "Hourly trips"
clock(ax, radii, title, "#dc143c")

plt.show()
```

In this code:

A polar subplot is created in a 3x3 grid (plt.subplot(3, 3, 1, polar=True)) as part of a larger figure.

The number of trips per hour is calculated from the 'pickup_hour' column in the DataFrame 'df'.

The clock function is then called to generate a polar bar plot (clock(ax, radii, title, "#dc143c")) with the calculated number of trips per hour.

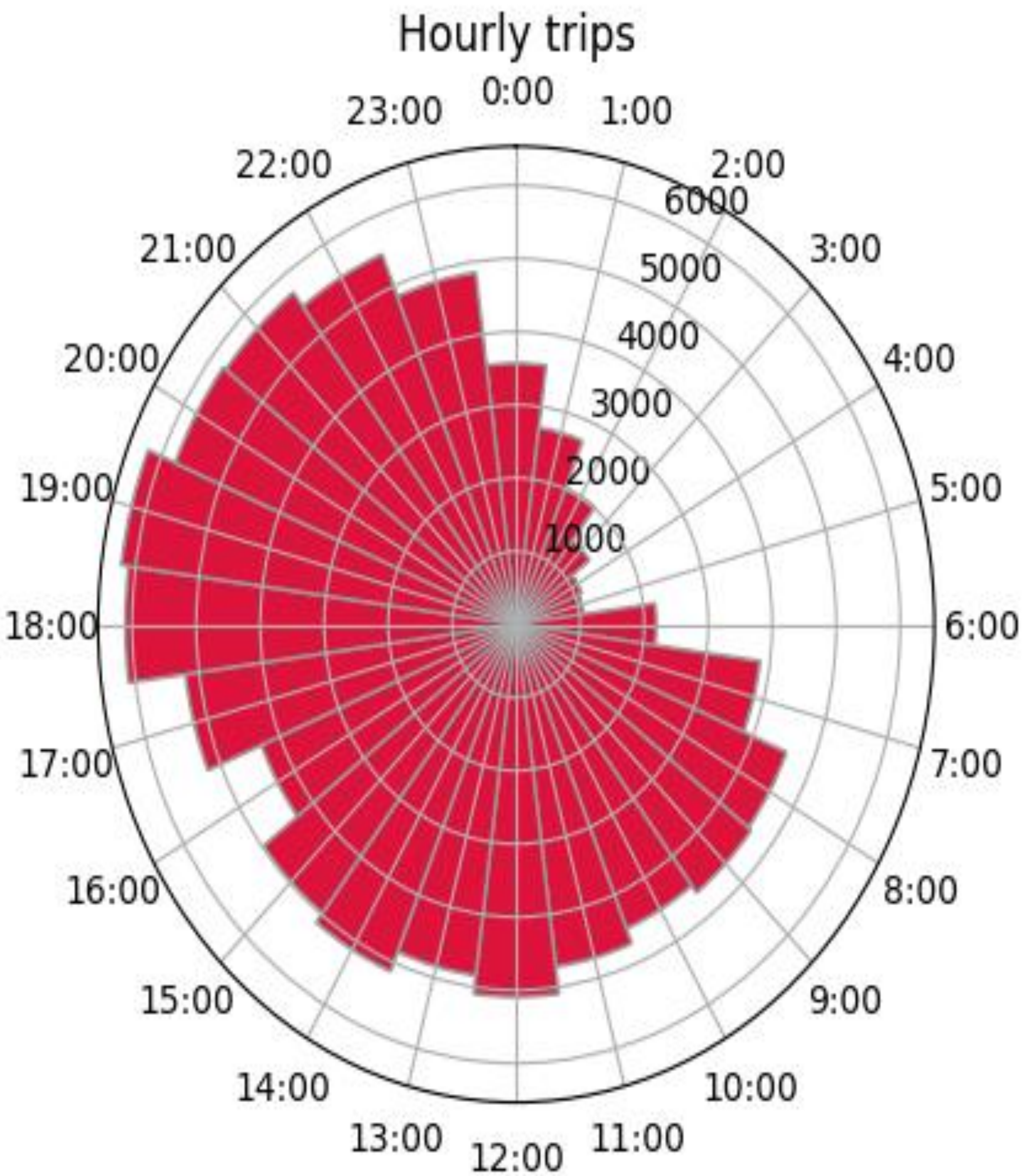
The resulting plot provides a visual representation of the hourly distribution of trips throughout the day.

The entire figure has a size of 15x15 inches (plt.figure(figsize=(15, 15))).

Finally, the plots are displayed using plt.show().

This code segment is useful for visualizing and analyzing the variation in trip counts across different hours of the day, offering insights into temporal patterns in the dataset.

➤ Output:-



```
[247] pip install folium
```

```
Requirement already satisfied: folium in /usr/local/lib/python3.10/dist-packages (0.14.0)
Requirement already satisfied: branca>=0.6.0 in /usr/local/lib/python3.10/dist-packages (from folium) (0.7.0)
Requirement already satisfied: jinja2>=2.9 in /usr/local/lib/python3.10/dist-packages (from folium) (3.1.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from folium) (1.23.5)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from folium) (2.31.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2>=2.9->fo
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from request
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->folium)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->fo
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->fo
```

In this section, we import the required libraries. **pandas** is a powerful data manipulation library in Python, and **folium** is a Python wrapper for Leaflet, a popular JavaScript library for creating interactive maps.

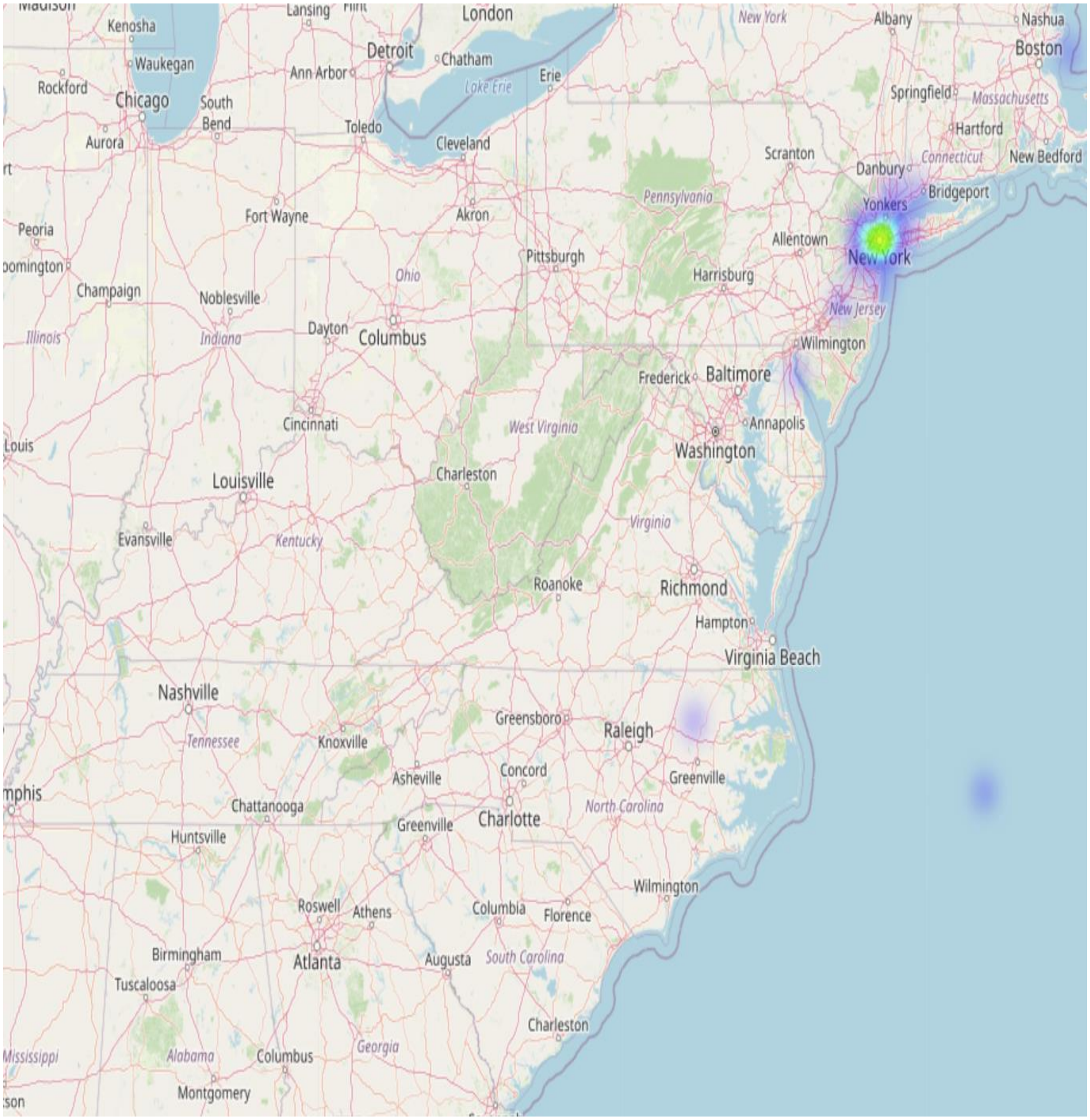
Here, we create a simple Pandas DataFrame with three columns: **latitude**, **longitude**, and **city**. This DataFrame represents some geographic data with latitude and longitude coordinates for three cities.

We create a base map using Folium. The **location** parameter is set to the mean of the latitude and longitude coordinates to center the map, and **zoom_start** sets the initial zoom level.

inally, we save the map as an HTML file using the **save** method. After running this script, you can open the generated **map.html** file in a web browser to visualize the map with the marked cities.

```
import folium
from folium.plugins import HeatMap
pickup_map = folium.Map(location=[df['pickup_latitude'].mean(), df['pickup_longitude'].mean()], zoom_start=12)
pickup_map.add_child(folium.plugins.HeatMap(df[['pickup_latitude', 'pickup_longitude']].values, radius=8))
pickup_map.save("pickup_Geo_map.html")
dropoff_map = folium.Map(location=[df['dropoff_latitude'].mean(), df['dropoff_longitude'].mean()], zoom_start=
dropoff_map.add_child(folium.plugins.HeatMap(df[['dropoff_latitude', 'dropoff_longitude']].values, radius=8))
dropoff_map.save("dropoff_Geo_map.html")
dropoff_map.save("dropoff_Geo_map.html")
```

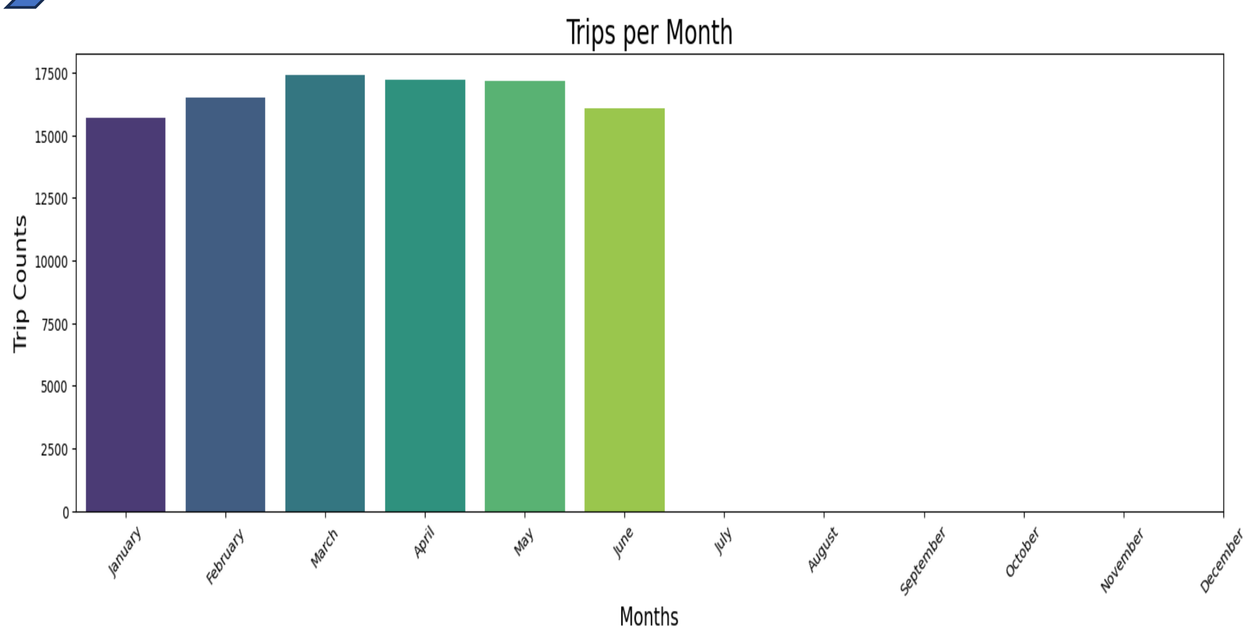
Output:-



Kenosha
Lansing
London
New York
Albany
Nashua
Waukegan
Detroit
Chatham
Boston
Ann Arbor
Lake Erie
Erie
Rockford
Chicago
South Bend
Springfield
Massachusetts
Toledo
Hartford
Aurora
Cleveland
Scranton
Connecticut
New Bedford
Danbury
Pennsylvania
Bridgeport
Yonkers
Fort Wayne
Akron
Peoria
Allentown
Pittsburgh
Harrisburg
Champaign
Noblesville
Ohio
New Jersey
Illinois
Indiana
Dayton
Columbus
Wilmington
Frederick
Baltimore
Cincinnati
Annapolis
West Virginia
Washington
Charleston
Louisville
Virginia
Evansville
Kentucky
Roanoke
Richmond
Hampton
Virginia Beach
Nashville
Greensboro
Raleigh
Tennessee
Knoxville
Greenville
Asheville
Concord
Charlotte
North Carolina
Chattanooga
Greenville
Huntsville
Wilmington
Roswell
Athens
Columbia
Florence
Birmingham
Atlanta
Augusta
South Carolina
Tuscaloosa
Charleston
Mississippi
Alabama
Columbus
Georgia
Montgomery

```python
# Create a figure and axes
plt.figure(figsize=(19, 5))

# Create a countplot with custom month labels
ax = sns.countplot(data=df, x='month', palette='viridis')

# Set axis labels and title
ax.set_ylabel('Trip Counts', fontsize=15)
ax.set_xlabel('Months', fontsize=15)
ax.set_title('Trips per Month', fontsize=20)

# Specify the positions and labels for the x-axis ticks
month_positions = list(range(12))
month_labels = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'Octobe

# Set the custom tick positions and labels for the x-axis
ax.set_xticks(month_positions)
ax.set_xticklabels(month_labels, rotation=45)

plt.show()
```

Output:-



The provided Python code utilizes Seaborn and Matplotlib to create a countplot visualizing the distribution of trips across different months. Here's a short note:

In this code:

A figure with a size of 19x5 inches is created using plt.figure(figsize=(19, 5)).

A countplot is generated using Seaborn (sns.countplot()) to display the number of trips per month based on the 'month' column in the DataFrame 'df'.

Axis labels ('Trip Counts' and 'Months') and a title ('Trips per Month') are set to enhance plot readability.

Custom labels for months are specified using the month_positions and month_labels variables.

The x-axis tick positions and labels are customized using ax.set_xticks() and ax.set_xticklabels() with rotation for better visibility.

The resulting plot provides a clear overview of trip counts distributed across the months of the year.

Finally, the plot is displayed using plt.show().

This code is effective for visualizing monthly variations in trip counts and allows for easy interpretation of temporal patterns in the dataset.

```python
plt.figure(figsize=(12, 4))


group1 = df.groupby('pickup_hour')['trip_duration'].mean().reset_index()

# Create a point plot
point_plot = sns.pointplot(x='pickup_hour', y='trip_duration', data=group1, color='skyblue')

point_plot.set_ylabel('Trip Duration (seconds)', fontsize=10)
point_plot.set_xlabel('Pickup Hour', fontsize=10)
point_plot.set_title('Trip Duration per Hour', fontsize=15)

plt.show()
```

Output:-

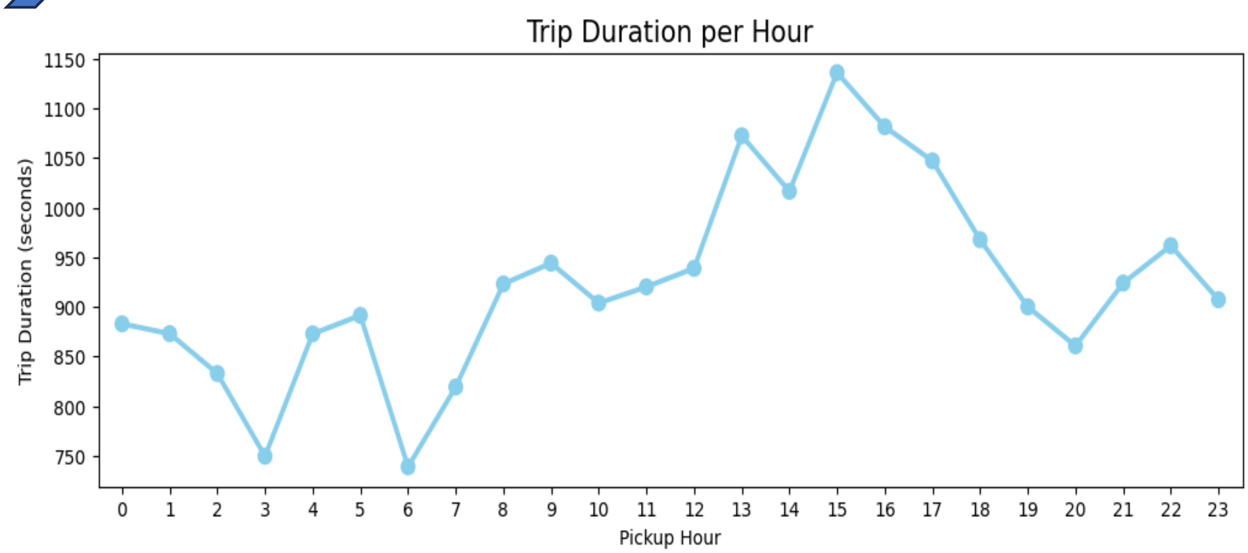


Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Grouping Data:

The DataFrame 'df' is grouped by 'pickup_hour,' and the mean trip duration for each hour is calculated using groupby('pickup_hour')['trip_duration'].mean().reset_index(). This creates a new DataFrame named 'group1' with columns 'pickup_hour' and 'trip_duration.'

Point Plot:

A point plot is created using Seaborn's sns.pointplot(). The x-axis represents the 'pickup_hour,' the y-axis represents the mean 'trip_duration,' and the data is taken from the 'group1' DataFrame.

The color of the points is set to 'skyblue.'

Axis Labels and Title:

Axis labels ('Pickup Hour' and 'Trip Duration (seconds)') and a title ('Trip Duration per Hour') are set using point_plot.set_ylabel(), point_plot.set_xlabel(), and point_plot.set_title() for better plot interpretation.

Display the Plot:

Finally, the plot is displayed using plt.show().

This code generates a point plot that visually represents the average trip duration for each hour, helping to identify trends or patterns in trip durations throughout the day. The 'groupby' operation is particularly useful for aggregating and summarizing data before visualization.

```
plt.figure(figsize=(12, 4))

# Group by weekday number and calculate the mean trip duration
group2 = df.groupby('weekday_num')['trip_duration'].mean().reset_index()

# Create a point plot
point_plot = sns.pointplot(x='weekday_num', y='trip_duration', data=group2, color='skyblue')

point_plot.set_ylabel('Trip Duration (seconds)', fontsize=10)
point_plot.set_xlabel('Weekday', fontsize=10)
point_plot.set_title('Trip Duration per Weekday', fontsize=15)

# Set x-axis labels to display the weekday names
weekday_labels = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
point_plot.set_xticklabels(weekday_labels, rotation=45)

plt.show()
```

Output:-

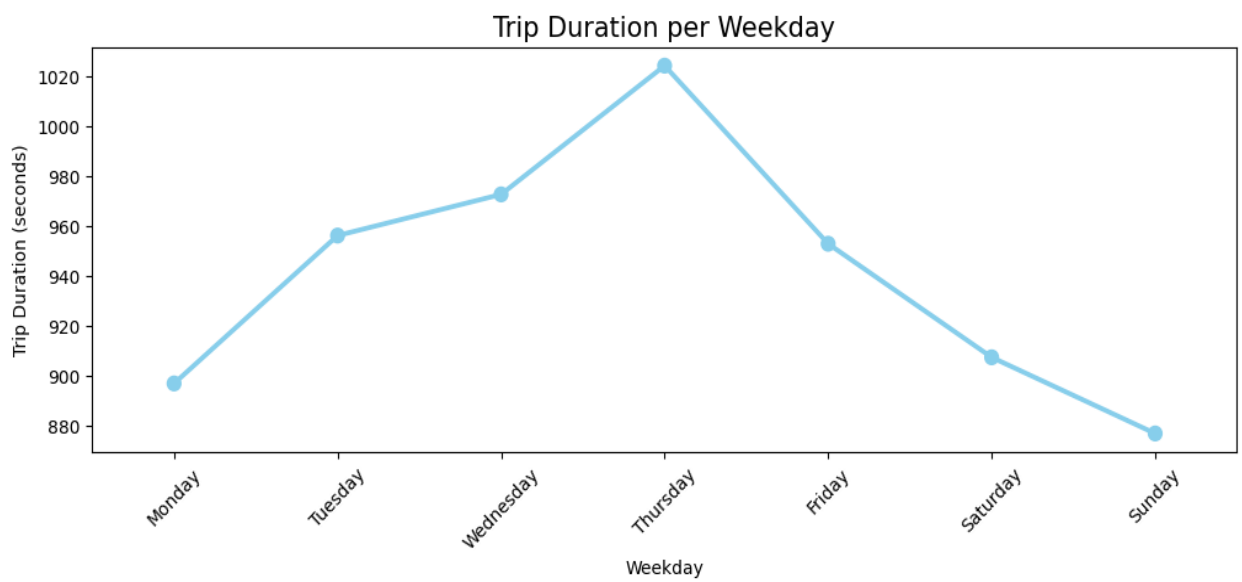


Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Grouping Data:

The DataFrame 'df' is grouped by 'weekday_num,' and the mean trip duration for each weekday is calculated using groupby('weekday_num')['trip_duration'].mean().reset_index(). This creates a new DataFrame named 'group2' with columns 'weekday_num' and 'trip_duration.'

Point Plot:

A point plot is created using Seaborn's sns.pointplot(). The x-axis represents the 'weekday_num,' the y-axis represents the mean 'trip_duration,' and the data is taken from the 'group2' DataFrame.

The color of the points is set to 'skyblue.'

Axis Labels and Title:

Axis labels ('Weekday' and 'Trip Duration (seconds)') and a title ('Trip Duration per Weekday') are set using point_plot.set_ylabel(), point_plot.set_xlabel(), and point_plot.set_title() for better plot interpretation.

Custom X-axis Labels:

The x-axis labels are set to display the names of weekdays instead of numeric codes using point_plot.set_xticklabels(weekday_labels, rotation=45).

Display the Plot:

Finally, the plot is displayed using plt.show().

This code generates a point plot that visually represents the average trip duration for each weekday, with custom weekday labels for improved readability. The 'groupby' operation facilitates the aggregation of data for visualization.

```
plt.figure(figsize=(12, 4))


group3 = df.groupby('month')['trip_duration'].mean().reset_index()

# Create a point plot
point_plot = sns.pointplot(x='month', y='trip_duration', data=group3, color='skyblue')

point_plot.set_ylabel('Trip Duration (seconds)', fontsize=10)
point_plot.set_xlabel('Month', fontsize=10)
point_plot.set_title('Trip Duration per Month', fontsize=15)

plt.show()
```

Output:-

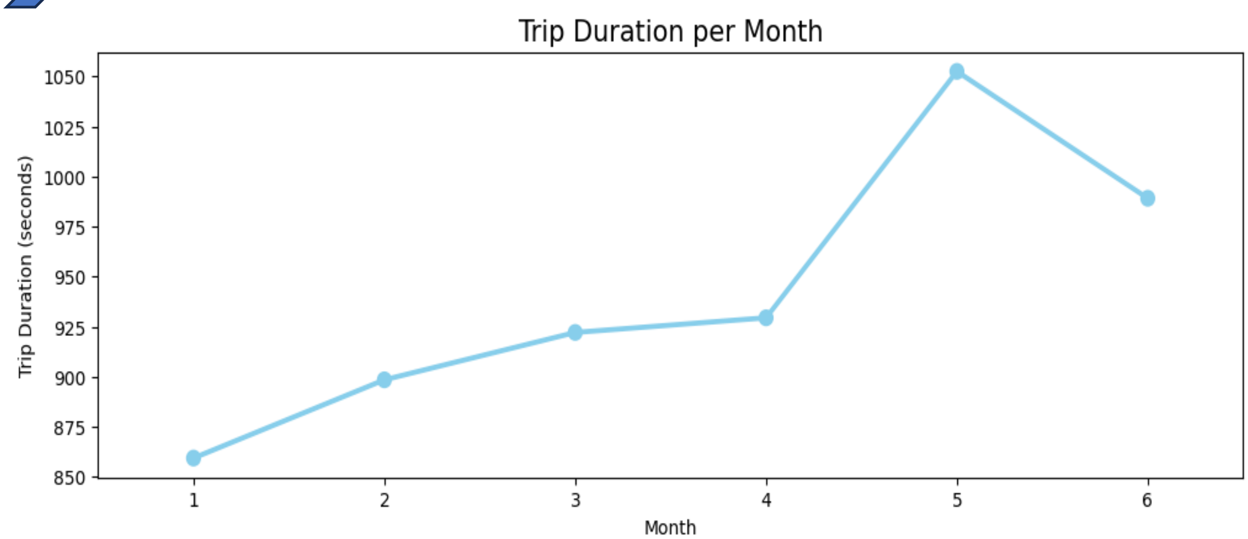


Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Grouping Data:

The DataFrame 'df' is grouped by 'month,' and the mean trip duration for each month is calculated using groupby('month')['trip_duration'].mean().reset_index(). This creates a new DataFrame named 'group3' with columns 'month' and 'trip_duration.'

Point Plot:

A point plot is created using Seaborn's sns.pointplot(). The x-axis represents the 'month,' the y-axis represents the mean 'trip_duration,' and the data is taken from the 'group3' DataFrame.

The color of the points is set to 'skyblue.'

Axis Labels and Title:

Axis labels ('Month' and 'Trip Duration (seconds)') and a title ('Trip Duration per Month') are set using point_plot.set_ylabel(), point_plot.set_xlabel(), and point_plot.set_title() for better plot interpretation.

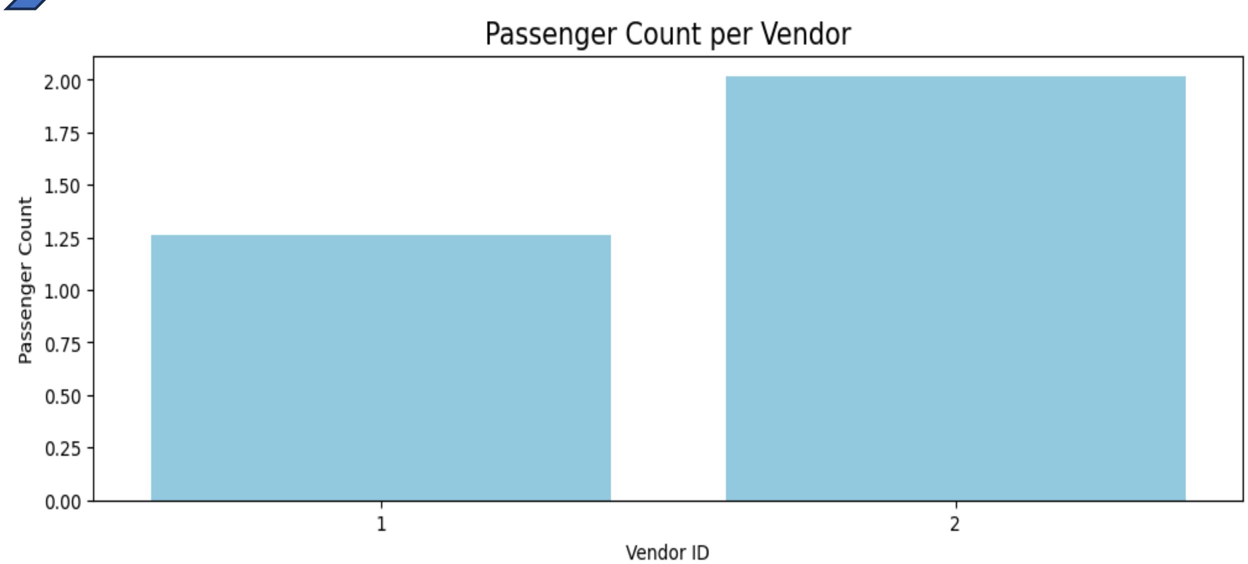
Display the Plot:

Finally, the plot is displayed using plt.show().

This code generates a point plot that visually represents the average trip duration for each month. The 'groupby' operation helps aggregate data by month, facilitating the visualization of temporal patterns in trip durations over the course of the year.

```python
group9 = df.groupby('vendor_id')['passenger_count'].mean().reset_index()

plt.figure(figsize=(12, 4))

# Create a bar plot
bar_plot = sns.barplot(x='vendor_id', y='passenger_count', data=group9, color='skyblue')

bar_plot.set_ylabel('Passenger Count', fontsize=10)
bar_plot.set_xlabel('Vendor ID', fontsize=10)
bar_plot.set_title('Passenger Count per Vendor', fontsize=15)

plt.show()
```

Output:-



Grouping Data:

The DataFrame 'df' is grouped by 'vendor_id,' and the mean passenger count for each vendor is calculated using groupby('vendor_id')['passenger_count'].mean().reset_index(). This creates a new DataFrame named 'group9' with columns 'vendor_id' and 'passenger_count.'

Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Bar Plot:

A bar plot is created using Seaborn's sns.barplot(). The x-axis represents the 'vendor_id,' the y-axis represents the mean 'passenger_count,' and the data is taken from the 'group9' DataFrame.

The color of the bars is set to 'skyblue.'

Axis Labels and Title:

Axis labels ('Vendor ID' and 'Passenger Count') and a title ('Passenger Count per Vendor') are set using bar_plot.set_ylabel(), bar_plot.set_xlabel(), and bar_plot.set_title() for better plot interpretation.

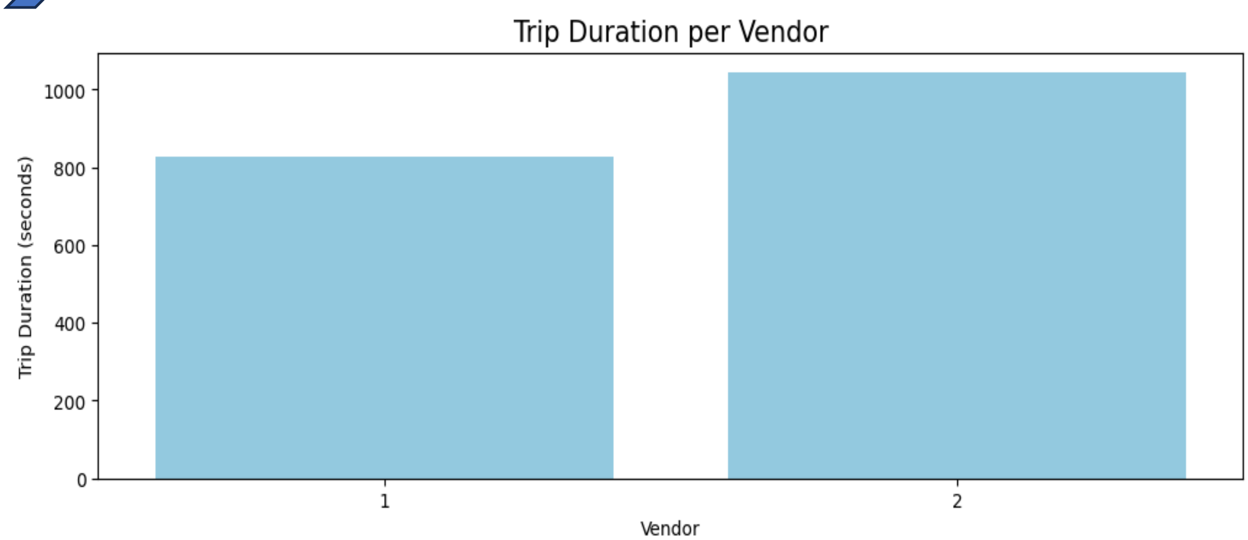
Display the Plot:

Finally, the bar plot is displayed using plt.show().

This code generates a bar plot that visually represents the average passenger count for each vendor. The 'groupby' operation helps aggregate data by vendor, facilitating the comparison of average passenger counts across different vendors in the dataset.

```python
group4 = df.groupby('vendor_id')['trip_duration'].mean().reset_index()

plt.figure(figsize=(12, 4))

# Create a bar plot
bar_plot = sns.barplot(x='vendor_id', y='trip_duration', data=group4, color='skyblue')

bar_plot.set_ylabel('Trip Duration (seconds)', fontsize=10)
bar_plot.set_xlabel('Vendor', fontsize=10)
bar_plot.set_title('Trip Duration per Vendor', fontsize=15)

plt.show()
```

Output:-



Grouping Data:

The DataFrame 'df' is grouped by 'vendor_id,' and the mean trip duration for each vendor is calculated using groupby('vendor_id')['trip_duration'].mean().reset_index(). This creates a new DataFrame named 'group4' with columns 'vendor_id' and 'trip_duration.'

Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Bar Plot:

A bar plot is created using Seaborn's sns.barplot(). The x-axis represents the 'vendor_id,' the y-axis represents the mean 'trip_duration,' and the data is taken from the 'group4' DataFrame.

The color of the bars is set to 'skyblue.'

Axis Labels and Title:

Axis labels ('Vendor' and 'Trip Duration (seconds)') and a title ('Trip Duration per Vendor') are set using bar_plot.set_ylabel(), bar_plot.set_xlabel(), and bar_plot.set_title() for better plot interpretation.

Display the Plot:

Finally, the bar plot is displayed using plt.show().

This code generates a bar plot that visually represents the average trip duration for each vendor. The 'groupby' operation helps aggregate data by vendor, facilitating the comparison of average trip durations across different vendors in the dataset.

```python
plt.figure(figsize=(12, 4))

# Create a countplot with custom x-axis labels
count_plot = sns.countplot(data=df, x='weekday_num', palette='viridis')

count_plot.set_xlabel('Weekday', fontsize=15)
count_plot.set_ylabel('Trip Counts', fontsize=15)
count_plot.set_title('Trips per Day', fontsize=20)

# Set the x-axis labels to display weekdays
weekday_labels = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
count_plot.set_xticklabels(weekday_labels, rotation=45)

plt.show()
```

Output:-

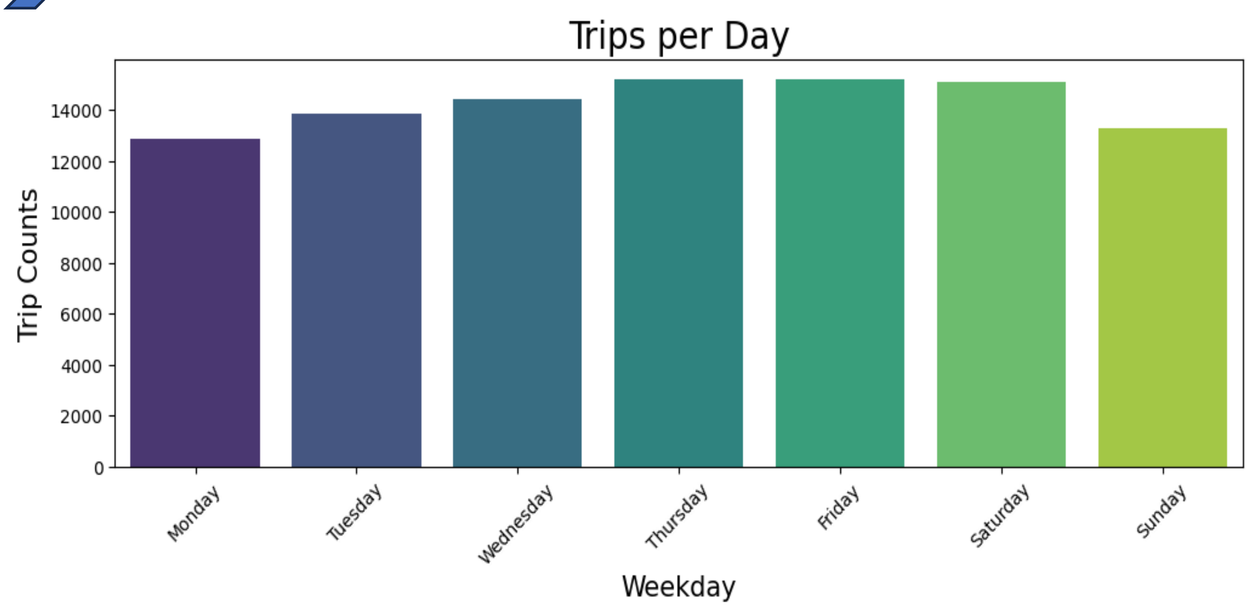


Figure Size:

A figure with a size of 12x4 inches is created using plt.figure(figsize=(12, 4)).

Count Plot:

A count plot is created using Seaborn's sns.countplot(). The x-axis represents the 'weekday_num,' and the count of trips for each weekday is displayed.

The color palette is set to 'viridis' for a visually appealing color scheme.

Axis Labels and Title:

Axis labels ('Weekday' and 'Trip Counts') and a title ('Trips per Day') are set using count_plot.set_xlabel(), count_plot.set_ylabel(), and count_plot.set_title() for better plot interpretation.

Custom X-axis Labels:

The x-axis labels are set to display the names of weekdays instead of numeric codes using count_plot.set_xticklabels(weekday_labels, rotation=45).

Display the Plot:

Finally, the count plot is displayed using plt.show().

This code generates a count plot that visually represents the distribution of trips across different weekdays. The custom x-axis labels enhance the interpretability of the plot by displaying the names of the weekdays. The 'countplot' is useful for understanding the frequency of trips on each day of the week.

```python
from sklearn.preprocessing import LabelEncoder

# Create a LabelEncoder object
label_encoder = LabelEncoder()

# Apply label encoding to the 'vendor_id' column and overwrite it
df['vendor_id'] = label_encoder.fit_transform(df['vendor_id'])
```

Certainly! This Python code snippet uses the scikit-learn library to perform label encoding on the 'vendor_id' column in a pandas DataFrame. Here's an explanation:

Import LabelEncoder:

The code begins by importing the LabelEncoder class from the sklearn.preprocessing module. This class is used for encoding categorical variables into numerical labels.

Create LabelEncoder Object:

An instance of the LabelEncoder class is created and assigned to the variable label_encoder.

Label Encoding:

The 'vendor_id' column in the DataFrame 'df' is selected, and the fit_transform method of the LabelEncoder is applied to it.

The fit_transform method fits the encoder to the unique values in the 'vendor_id' column and transforms these categorical values into numerical labels.

The original 'vendor_id' column is then overwritten with the newly encoded values.

Explanation:

Label encoding is a technique used to convert categorical data (like 'vendor_id') into numerical format, which is often required for machine learning algorithms.

The LabelEncoder is fit to the unique values in the 'vendor_id' column using fit_transform.

The transformed numerical labels replace the original categorical values in the 'vendor_id' column.

The fit_transform method is used in this case, which is a combination of fitting the encoder to the unique values and transforming them simultaneously.

By executing this code, the 'vendor_id' column is transformed from categorical labels (e.g., 'V1', 'V2') to numerical labels (e.g., 0, 1), making the data suitable for certain machine learning algorithms that require numerical inputs.

df.dtypes

```
df.dtypes
```

```
id                              object
vendor_id                        int64
pickup_datetime         datetime64[ns]
dropoff_datetime        datetime64[ns]
passenger_count                  int64
pickup_longitude               float64
pickup_latitude                float64
dropoff_longitude              float64
dropoff_latitude               float64
trip_duration                    int64
month                            int32
weekday                         object
weekday_num                      int32
pickup_hour                      int32
store_and_fwd_flag_Y              bool
dtype: object
```

df.head()

Output:-

```
df.head()
```

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latitu
209174	id2104500	1	2016-03-09 20:09:32	2016-03-09 20:33:18	5	-74.00	40.
1296156	id1063788	1	2016-05-26 08:38:50	2016-05-26 08:51:19	1	-73.97	40.
136770	id1376921	0	2016-06-12 10:36:15	2016-06-12 10:50:21	1	-73.99	40.
219673	id1418169	1	2016-06-26 02:13:21	2016-06-26 02:16:28	1	-73.98	40.
218117	id2470055	1	2016-02-17 11:33:38	2016-02-17 11:39:45	1	-73.98	40.

```
df.head()
```

dropoff_longitude	dropoff_latitude	trip_duration	month	weekday	weekday_num	pickup_hour	store_and_fwd_fla
-73.96	40.67	1426	3	Wednesday	2	20	F
-73.99	40.75	749	5	Thursday	3	8	F
-74.00	40.71	846	6	Sunday	6	10	F
-73.98	40.73	187	6	Sunday	6	2	F
-73.99	40.73	367	2	Wednesday	2	11	F

```
[33] from sklearn.preprocessing import MinMaxScaler


     scaler = MinMaxScaler()
     normalizing_column = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude']
     df[normalizing_column] = scaler.fit_transform(df[normalizing_column])
```

Explanation:

Normalization is a preprocessing step that scales numerical features to a specific range, often [0, 1], to prevent any feature from dominating others.

The MinMaxScaler is fit to the selected columns ('pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude') using fit_transform.

The transformed values replace the original values in the specified columns in the DataFrame 'df'.

Normalization is particularly useful for machine learning algorithms that are sensitive to the scale of input features.

By executing this code, the specified columns in the DataFrame 'df' are normalized, ensuring that their values are within the [0, 1] range.

```
from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Define the columns to be standardized
columns_to_standardize = ['trip_duration']

# Apply standardization to the selected columns
df[columns_to_standardize] = scaler.fit_transform(df[columns_to_standardize])
```

Explanation:

Standardization is a preprocessing step that scales numerical features to have zero mean and unit variance.

The StandardScaler is fit to the selected column ('trip_duration') using fit_transform.

The transformed values replace the original values in the specified column in the DataFrame 'df'.

Standardization is particularly useful for machine learning algorithms that assume features are normally distributed and have similar scales.

By executing this code, the specified column in the DataFrame 'df' is standardized, ensuring that its values have a mean of 0 and a standard deviation of 1.

Standardization:

The specified column in the DataFrame 'df' is selected, and the fit_transform method of the StandardScaler is applied to it.

The fit_transform method standardizes the selected column by removing the mean and scaling to unit variance.

The original values in the specified column are replaced with their standardized counterparts.

Import StandardScaler:

The code begins by importing the StandardScaler class from the sklearn.preprocessing module. This class is used for standardizing numerical features by removing the mean and scaling to unit variance.

```
from scipy import stats

# Select the numerical columns for outlier detection
numerical_columns = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'trip

# Calculate the absolute Z-scores for each numerical column
z_scores = np.abs(stats.zscore(df[numerical_columns]))

# Define a threshold for considering data points as outliers (adjust as needed)
z_threshold = 3.0

# Find rows with outliers
outliers = (z_scores > z_threshold).any(axis=1)

# Remove rows with outliers
df_clean = df[~outliers]

# Display the cleaned DataFrame
df_clean.head()

df = df_clean

# Display the cleaned DataFrame
df.head()
```

Output:-

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_lati
209174	id2104500	1	2016-03-09 20:09:32	2016-03-09 20:33:18	5	0.50	
1296156	id1063788	1	2016-05-26 08:38:50	2016-05-26 08:51:19	1	0.50	
136770	id1376921	0	2016-06-12 10:36:15	2016-06-12 10:50:21	1	0.50	
219673	id1418169	1	2016-06-26 02:13:21	2016-06-26 02:16:28	1	0.50	
218117	id2470055	1	2016-02-17 11:33:38	2016-02-17 11:39:45	1	0.50	

longitude	dropoff_latitude	store_and_fwd_flag	trip_duration	month	weekday	weekday_num	pickup_hour
0.62	0.63	N	0.16	3	Wednesday	2	20
0.61	0.64	N	-0.06	5	Thursday	3	8
0.61	0.63	N	-0.03	6	Sunday	6	10
0.62	0.64	N	-0.25	6	Sunday	6	2
0.61	0.64	N	-0.19	2	Wednesday	2	11

Import Required Libraries:

The code begins by importing the necessary libraries, including scipy.stats for statistical functions.

from scipy import stats

Select Numerical Columns:

A list named numerical_columns is created, containing the names of the numerical columns for which outliers will be detected. In this case, it includes 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', and 'trip_duration'.

numerical_columns = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude', 'dropoff_latitude', 'trip_duration']

Calculate Z-scores:

The absolute Z-scores for each numerical column are calculated using the stats.zscore function from the scipy library. Z-scores measure how many standard deviations away each data point is from the mean.

```
z_scores = np.abs(stats.zscore(df[numerical_columns]))
```

Set Z-score Threshold:

A threshold for considering data points as outliers is defined. In this case, a threshold of 3.0 is set (adjustable based on specific requirements).

```
z_threshold = 3.0
```

Detect Outliers:

Rows with outliers are identified by checking if the absolute Z-scores exceed the defined threshold. The result is a boolean array.

```
outliers = (z_scores > z_threshold).any(axis=1)
```

Remove Outliers:

Rows with outliers are removed from the original DataFrame using boolean indexing.

```
df_clean = df[~outliers]
```

Update Original DataFrame:

The original DataFrame 'df' is updated with the cleaned DataFrame 'df_clean'.

```
df = df_clean
```

Display Cleaned DataFrame:

The cleaned DataFrame is displayed to inspect the results.

```
df.head()
```

By executing this code, outliers beyond the specified Z-score threshold are identified and removed from the specified numerical columns, resulting in a cleaned DataFrame. Adjustments to the threshold can be made based on the specific characteristics of the data.

df.shape

```
df.shape
```

```
(95950, 15)
```

```
# Select only the numeric columns in your DataFrame
df2_matrix_columns = df.select_dtypes(include=['number'])
df2_matrix = df2_matrix_columns.corr()

# Creating a heatmap with a different color palette ('viridis' in this case)
plt.figure(figsize=(10, 8))
sns.heatmap(df2_matrix, annot=True, cmap='viridis', linewidths=0.7)
plt.title('Heatmap')
plt.show()
```

Select Numeric Columns:

The code begins by selecting only the numeric columns from the DataFrame 'df' using select_dtypes(include=['number']). This ensures that only columns with numerical data are included in the correlation analysis.

df2_matrix_columns = df.select_dtypes(include=['number'])

Calculate Correlation Matrix:

The correlation matrix is calculated using the corr() method on the selected numeric columns.

df2_matrix = df2_matrix_columns.corr()

Create Heatmap:

A heatmap is created using Seaborn's sns.heatmap() function.

The correlation matrix is passed as input to the data parameter.

annot=True adds numeric annotations to the heatmap cells, showing the correlation values.

cmap='viridis' sets the color palette for the heatmap to 'viridis'.

linewidths=0.7 controls the width of the lines between cells in the heatmap.

plt.figure(figsize=(10, 8))

sns.heatmap(df2_matrix, annot=True, cmap='viridis', linewidths=0.7)

Set Title and Show the Plot:

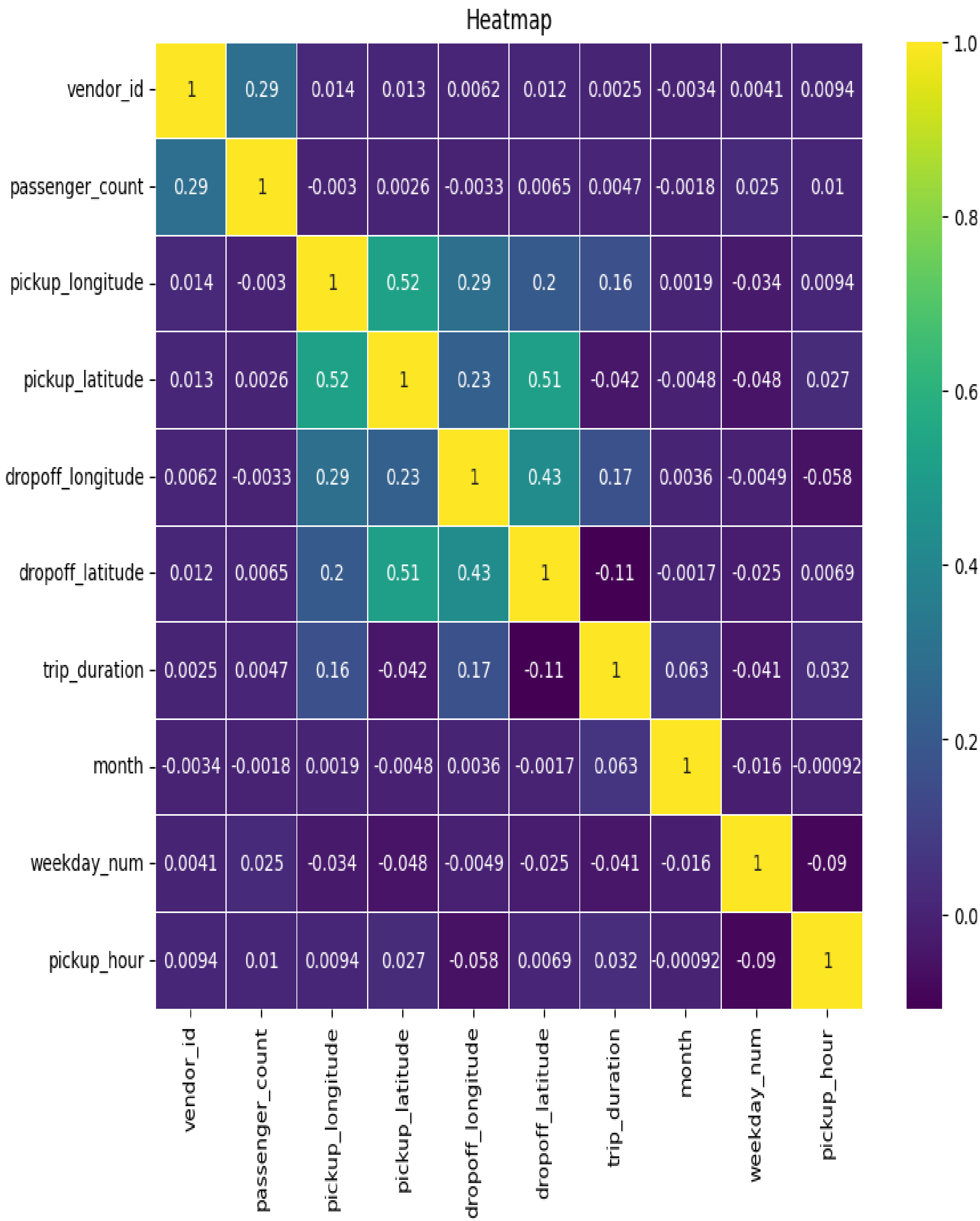
A title 'Heatmap' is set for the heatmap using plt.title('Heatmap').

Finally, the plot is displayed using plt.show().

```
plt.title('Heatmap')
plt.show()
```

By executing this code, you'll get a heatmap that visually represents the correlation between different numeric columns in the DataFrame. Darker colors indicate stronger correlations, while lighter colors indicate weaker or no correlations. The annotations within the cells provide the exact correlation values. This visualization is useful for understanding relationships between variables in the dataset.

Output:-



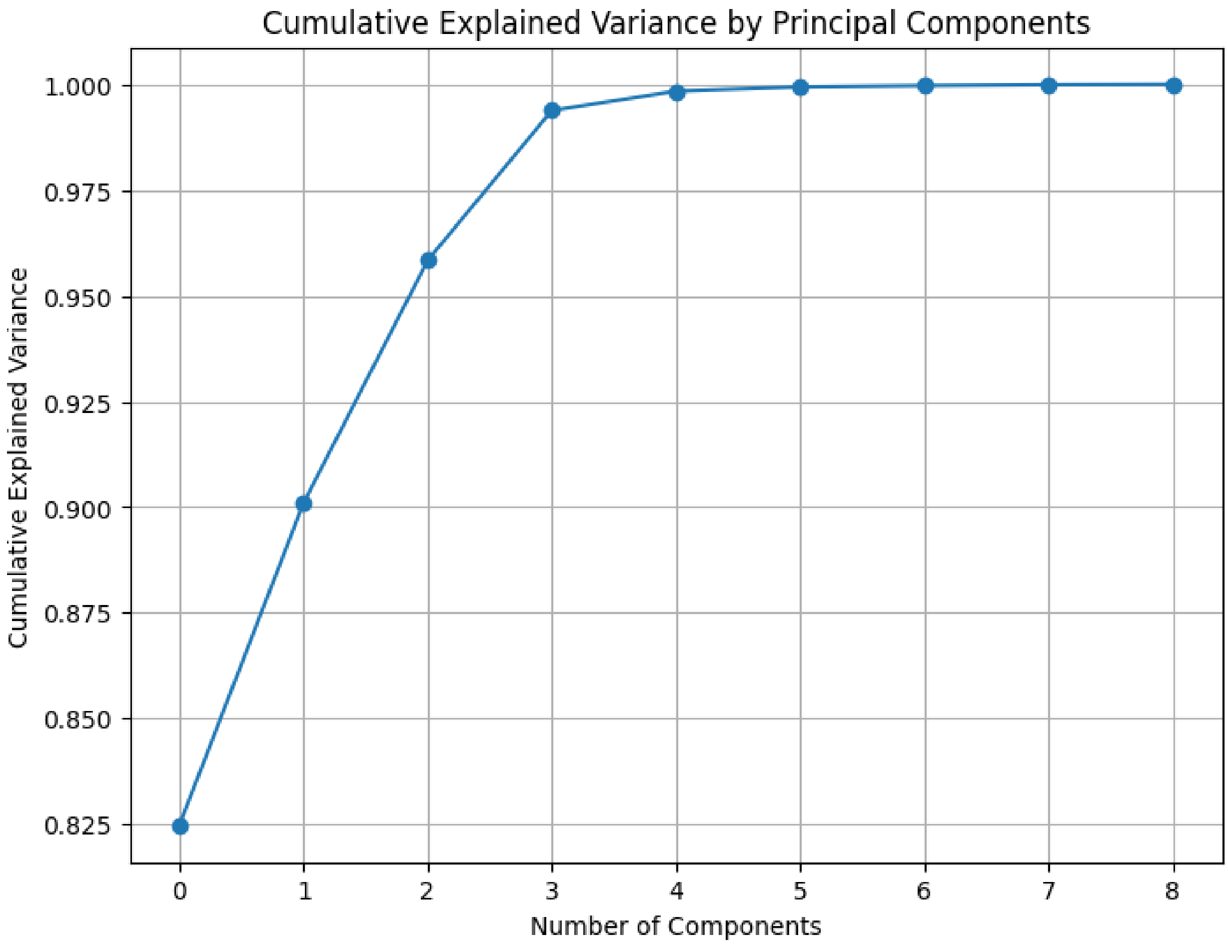
Heatmap
vendor_id
passenger_count
pickup_longitude
pickup_latitude
dropoff_longitude
dropoff_latitude
trip_duration
month
weekday_num
pickup_hour
1 0.29 0.014 0.013 0.0062 0.012 0.0025 -0.0034 0.0041 0.0094
0.29 1 -0.003 0.0026 -0.0033 0.0065 0.0047 -0.0018 0.025 0.01
0.014 -0.003 1 0.52 0.29 0.2 0.16 0.0019 -0.034 0.0094
0.013 0.0026 0.52 1 0.23 0.51 -0.042 -0.0048 -0.048 0.027
0.0062 -0.0033 0.29 0.23 1 0.43 0.17 0.0036 -0.0049 -0.058
0.012 0.0065 0.2 0.51 0.43 1 -0.11 -0.0017 -0.025 0.0069
0.0025 0.0047 0.16 -0.042 0.17 -0.11 1 0.063 -0.041 0.032
-0.0034 -0.0018 0.0019 -0.0048 0.0036 -0.0017 0.063 1 -0.016 -0.00092
0.0041 0.025 -0.034 -0.048 -0.0049 -0.025 -0.041 -0.016 1 -0.09
0.0094 0.01 0.0094 0.027 -0.058 0.0069 0.032 -0.00092 -0.09 1
1.0
0.8
0.6
0.4
0.2
0.0

```
#X = df[predictors]
#y = df[target]
#y = df.iloc[:,9].values
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7294)
```

In this code we are commenting every thing (#).

```
from sklearn.decomposition import PCA

# Fit PCA to your training data
pca = PCA()
pca.fit(X_train)

# Calculate the cumulative explained variance
cumulative_explained_variance = np.cumsum(pca.explained_variance_ratio_)

# Create a plot to visualize the cumulative explained variance
plt.figure(figsize=(8, 6))
plt.plot(cumulative_explained_variance, marker='o')
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("Cumulative Explained Variance by Principal Components")
plt.grid()
plt.show()
```

Output:-



Import Necessary Libraries:

Import the required libraries, including PCA from scikit-learn and Matplotlib for plotting.

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

Fit PCA to Training Data:

Create a PCA instance (pca) and fit it to the training data (X_train).

pca = PCA()

pca.fit(X_train)

Calculate Cumulative Explained Variance:

Retrieve the explained variance ratios for each principal component and calculate the cumulative explained variance.

cumulative_explained_variance = np.cumsum(pca.explained_variance_ratio_)

Create Plot:

Create a line plot to visualize the cumulative explained variance. The x-axis represents the number of principal components, and the y-axis represents the cumulative explained variance.

plt.figure(figsize=(8, 6))

plt.plot(cumulative_explained_variance, marker='o')

Add Labels and Title:

Add labels to the axes and a title to the plot for better interpretation.

plt.xlabel("Number of Components")

plt.ylabel("Cumulative Explained Variance")

plt.title("Cumulative Explained Variance by Principal Components")

Display Grid and Show Plot:

Add a grid to the plot for better readability and display the plot.

plt.grid()

plt.show()

This code provides a visual representation of how much variance in the original features is explained by an increasing number of principal components. The plot helps in deciding how many principal components to retain in order to capture a significant amount of variance in the data.

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import  r2_score



predictors = ['vendor_id', 'passenger_count', 'pickup_longitude', 'pickup_latitude', 'dropoff_longitude
target = 'trip_duration'


X = df[predictors]
y = df.iloc[:,9].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7294)


#x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,random_state = 7294)

# Creating a linear regression model
model = LinearRegression()


model.fit(X_train, y_train)


y_pred = model.predict(X_test)


r_2 = r2_score(y_test, y_pred)
```

```
[150] r_2 = r2_score(y_test, y_pred)


      coefficients = dict(zip(predictors, model.coef_))
      print(" Coefficients:")
      print(coefficients)


      print("R-squared (R2) Score:", r_2)
```

```
 Coefficients:
{'vendor_id': 0.0002816130547943861, 'passenger_count': 0.001164609792535729, 'pickup_longitude': 9.07986098049
R-squared (R2) Score: 0.09122909659552003
```

Output:-


```
 Coefficients:
{'vendor_id': 0.0002816130547943861, 'passenger_count': 0.001164609792535729, 'pickup_longitude': 9.07986098049
R-squared (R2) Score: 0.09122909659552003
```


Define Predictors and Target:

Declare a list of feature column names (predictors) and the target variable name (target).

Split Data into Training and Testing Sets:

Use train_test_split to split the data into training and testing sets.

Create and Train a Linear Regression Model:

Instantiate a linear regression model (model) and fit it to the training data (X_train and y_train).

Make Predictions and Evaluate R-squared:

Predict target values (y_pred) using the trained model on the test set (X_test).

Calculate the R-squared (coefficient of determination) score using r2_score between the predicted and actual target values.

Print Coefficients and R-squared Score:

Display the coefficients of the linear regression model for each predictor.

Print the R-squared score, which represents the proportion of the variance in the target variable explained by the model.

```
cumulative_variance = np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4) * 100)
explained_variance = list(zip(range(1, len(cumulative_variance) + 1), cumulative_variance))
print(explained_variance)
```

```
[(1, 82.48), (2, 90.10000000000001), (3, 95.84), (4, 99.39), (5, 99.85), (6, 99.94999999999999), (7, 99.9799999
```

```
[152] from sklearn.tree import DecisionTreeRegressor

      # Create and fit the Decision Tree regression model, measuring the time
      start_time = time.time()
      dt_regression = DecisionTreeRegressor().fit(X_train, y_train)
      end_time = time.time()
      dt_time = end_time - start_time

      print(f"Time to train Decision Tree model: {dt_time:.2f} seconds")

      # Predict on the test data
      trips = dt_regression.predict(X_test)
```

Output:- Time to train Decision Tree model: 1.18 seconds

```
[153] de_t_score = r2_score(y_test, trips)
      print(de_t_score)
```

```
0.37492019030101775
```

Output:- 0.37492019030101775

```python
from sortedcontainers import SortedDict
sorted_trip_data = SortedDict()
for index, row in df.iterrows():
    tr_id = row['id']
    trip_duration = row['trip_duration']
    sorted_trip_data[trip_duration] = (tr_id, row)
new_trp_id = 'new_trp_id'
new_tr_duration = 500
new_tr_data = {
    'vendor_id': 'new_vendor',
    'pickup_datetime': '2023-10-28 10:00:00',
    'dropoff_datetime': '2023-10-28 10:30:00',
    'passenger_count': 3,
    'pickup_longitude': -73.9895,
    'pickup_latitude': 40.7523,
    'dropoff_longitude': -73.9876,
    'dropoff_latitude': 40.7612,
    'store_and_fwd_flag': 'N',
}
sorted_trip_data[new_tr_duration] = (new_trp_id, new_tr_data)
for duration, (tr_id, data) in sorted_trip_data.items():
    print(f"Trip ID: {tr_id}, Duration: {duration} seconds")
```

```
Trip ID: id3712815, Duration: -0.30631918542086234 seconds
Trip ID: id0771057, Duration: -0.3059936421418518 seconds
Trip ID: id2899379, Duration: -0.3056680988628413 seconds
Trip ID: id3499191, Duration: -0.3053425555838308 seconds
Trip ID: id0048237, Duration: -0.30501701230482026 seconds
Trip ID: id2621196, Duration: -0.30469146902580974 seconds
Trip ID: id1840822, Duration: -0.3043659257467992 seconds
Trip ID: id2589768, Duration: -0.30404038246778875 seconds
Trip ID: id2841671, Duration: -0.30371483918877823 seconds
Trip ID: id1550403, Duration: -0.303389295909767 seconds
Trip ID: id2431464, Duration: -0.3030637526307572 seconds
Trip ID: id0842166, Duration: -0.30273820935174667 seconds
Trip ID: id0430308, Duration: -0.30241266607273615 seconds
Trip ID: id1021500, Duration: -0.3020871227937256 seconds
Trip ID: id0761104, Duration: -0.3017615795147151 seconds
Trip ID: id0925860, Duration: -0.3014360362357046 seconds
Trip ID: id1839162, Duration: -0.30111049295669406 seconds
Trip ID: id0512802, Duration: -0.3007849496776836 seconds
Trip ID: id2864109, Duration: -0.3004594063986731 seconds
Trip ID: id3433450, Duration: -0.30013386311966256 seconds
Trip ID: id1487168, Duration: -0.29980831984065204 seconds
Trip ID: id0889412, Duration: -0.2994827765616415 seconds
Trip ID: id1729290, Duration: -0.299157233282631 seconds
Trip ID: id2284694, Duration: -0.2988316900036205 seconds
Trip ID: id3294655, Duration: -0.29850614672460996 seconds
```

```
Trip ID: id0834441, Duration: 1.2823320161504512 seconds
Trip ID: id1147522, Duration: 1.2953537473108718 seconds
Trip ID: id2550439, Duration: 1.3051200456811873 seconds
Trip ID: id1009342, Duration: 1.334093397513123 seconds
Trip ID: id1622983, Duration: 1.354277080811775 seconds
Trip ID: id0243519, Duration: 1.3549281673697962 seconds
Trip ID: id1480601, Duration: 1.3679498985302168 seconds
Trip ID: id1395535, Duration: 1.3806460864116268 seconds
Trip ID: id2016641, Duration: 1.3933422742930368 seconds
Trip ID: id2180879, Duration: 1.4047362890584048 seconds
Trip ID: id3166949, Duration: 1.4167813903817938 seconds
Trip ID: id2715742, Duration: 1.4190601933348674 seconds
Trip ID: id0321011, Duration: 1.4255710589150778 seconds
Trip ID: id2253572, Duration: 1.4398949631915405 seconds
Trip ID: id3978400, Duration: 1.448359088445814 seconds
Trip ID: id0706029, Duration: 1.4600786464901925 seconds
Trip ID: id2455471, Duration: 1.4613808196062346 seconds
Trip ID: id0598906, Duration: 1.5072824219467171 seconds
Trip ID: id2599873, Duration: 1.5124911144108855 seconds
Trip ID: id3806679, Duration: 1.514118830805938 seconds
Trip ID: id1452618, Duration: 1.5714144479117886 seconds
Trip ID: id0998434, Duration: 1.602015516138777 seconds
Trip ID: id3300797, Duration: 1.6072242086029453 seconds
Trip ID: id2806945, Duration: 1.655730157175512 seconds
Trip ID: id1424187, Duration: 1.7533931408786665 seconds
Trip ID: id3799848, Duration: 1.8836104524828725 seconds
Trip ID: id0413474, Duration: 1.92625662203325 seconds
Trip ID: id2456201, Duration: 2.3943878572503707 seconds
Trip ID: new_trp_id, Duration: 500 seconds
```

Explain which data structure was selected for each of the following cases (from arrays, stacks, queues, linked lists, and hash tables) to hold the data:

• The dataset ought to be displayed and sorted according to trip time in ascending order. Fresh information is

regularly added to the dataset, where these new journeys ought to appear at the conclusion of the sorted list.

• Example 1: Data Sorting and Data Viewing in Trip Duration Ascending Order

• Array data structure

• Justification: Numerical data, such as trip durations, can be efficiently stored and sorted using arrays. Those

allow for easy access to sorted data, making it perfect for viewing in ascending order.

Option 2: Easily Sorting Journeys based on Passenger Phone Numbers

• Dictionary-based hash tables as a data structure

• Rationale: Hash tables are excellent for quickly retrieving data using keys.

Using trip data as values and numbers as keys enables blazing-fast filtering, making them

the best option for this assignment

```python
import random

random_phone_numbers = [f"{random.randint(100, 999)}-{random.randint(100, 999)}-{random.randint(1000, 9999)}"

# Adding the 'phone_number' column to the DataFrame
df['phone_number'] = random_phone_numbers


print(df)
```

```
                id  vendor_id     pickup_datetime    dropoff_datetime  \
209174   id2104500          1 2016-03-09 20:09:32 2016-03-09 20:33:18
1296156  id1063788          1 2016-05-26 08:38:50 2016-05-26 08:51:19
136770   id1376921          0 2016-06-12 10:36:15 2016-06-12 10:50:21
219673   id1418169          1 2016-06-26 02:13:21 2016-06-26 02:16:28
218117   id2470055          1 2016-02-17 11:33:38 2016-02-17 11:39:45
...            ...        ...                 ...                 ...
1457328  id0584362          0 2016-06-23 11:39:08 2016-06-23 12:12:10
471121   id3134223          0 2016-04-16 01:34:03 2016-04-16 01:37:55
527507   id3428610          1 2016-02-04 21:37:44 2016-02-04 21:43:00
260392   id0320852          0 2016-02-28 20:20:10 2016-02-28 20:24:17
1151751  id3620140          0 2016-02-09 22:51:56 2016-02-09 23:06:14

         passenger_count  pickup_longitude  pickup_latitude  \
209174                 5              0.50             0.76
1296156                1              0.50             0.76
136770                 1              0.50             0.76
219673                 1              0.50             0.76
218117                 1              0.50             0.76
...                  ...               ...              ...
1457328                1              0.50             0.76
471121                 1              0.50             0.76
527507                 5              0.50             0.76
260392                 1              0.50             0.76
1151751                2              0.50             0.76

         dropoff_longitude  dropoff_latitude  trip_duration  month    weekday  \
209174                0.62              0.63           0.16      3  Wednesday
1296156               0.61              0.64          -0.06      5   Thursday
```

Output:-

Generate Random Phone Numbers:

Use a list comprehension to create a list of random phone numbers in the format "XXX-XXX-XXXX", where X is a random digit.

Create 'phone_number' Column:

Add a new column named 'phone_number' to the DataFrame (df) and assign the generated random phone numbers.

Display the DataFrame:

Print or display the DataFrame to view the added 'phone_number' column.

This code demonstrates how to generate and add random phone numbers to a pandas DataFrame, which can be useful for creating synthetic data or testing scenarios where phone numbers are needed.

```
# Check the random phone numbers for the first 3 rows of data
for i in range(3):
    row = df.iloc[i]
    phone_number = row['phone_number']
    print(f"Row {i + 1} - Phone Number: {phone_number}")
```

Output:-

Row 1 - Phone Number: 846-908-8038
Row 2 - Phone Number: 766-942-9603
Row 3 - Phone Number: 914-884-8688

Here I am displaying the 3 phone numbers.

df.head()

Output:-

	id	vendor_id	pickup_datetime	dropoff_datetime	passenger_count	pickup_longitude	pickup_latit
209174	id2104500	1	2016-03-09 20:09:32	2016-03-09 20:33:18	5	0.50	(
1296156	id1063788	1	2016-05-26 08:38:50	2016-05-26 08:51:19	1	0.50	(
136770	id1376921	0	2016-06-12 10:36:15	2016-06-12 10:50:21	1	0.50	(
219673	id1418169	1	2016-06-26 02:13:21	2016-06-26 02:16:28	1	0.50	(
218117	id2470055	1	2016-02-17 11:33:38	2016-02-17 11:39:45	1	0.50	(

pickup_latitude	dropoff_longitude	dropoff_latitude	trip_duration	month	weekday	weekday_num	pickup_hour
0.76	0.62	0.63	0.16	3	Wednesday	2	20
0.76	0.61	0.64	-0.06	5	Thursday	3	8
0.76	0.61	0.63	-0.03	6	Sunday	6	10
0.76	0.62	0.64	-0.25	6	Sunday	6	2
0.76	0.61	0.64	-0.19	2	Wednesday	2	11

ff_latitude	trip_duration	month	weekday	weekday_num	pickup_hour	store_and_fwd_flag_Y	phone_number
0.63	0.16	3	Wednesday	2	20	0	846-908-8038
0.64	-0.06	5	Thursday	3	8	0	766-942-9603
0.63	-0.03	6	Sunday	6	10	0	914-884-8688
0.64	-0.25	6	Sunday	6	2	0	929-809-5830
0.64	-0.19	2	Wednesday	2	11	0	356-951-1529

4.Algorithm and Data Structure Efficiency:

Explaining one more method by typical mode in addition .

Linked List or Doubly Linked List Data Structure

Reasoning:

When it's required to make frequent additions and keep the list in sorted order, linked lists work well. Since new trips are added regularly in this circumstance, they must be positioned at the end of the sorted list.

O(1) time complexity linked lists provide efficient insertion at the tail.

A double linked list leads to O(n) time complexity for sorting, but makes sorting by trip duration easier by rearranging the connections without requiring a change to the entire list.

effective at keeping the sorted order in place even when additions happen frequently.

Hash Table (HashMap) as a Data Structure

Reasoning:

Hash tables offer fast access based on keys (the passenger's phone number in this example), and their insertion, deletion, and search operations have an average-case time complexity of O(1).

A distinct identity or key is implied by adding a new column for the passenger's phone number, making it perfect for hash table lookup.

With the use of a phone number as a key, hash tables provide effective retrieval of individual passenger's flights.

They offer quick key-based filtering capabilities, which makes it effective for removing journeys taken by a particular passenger.