Business Case: Delhivery - Feature Engineering

Goal of the Project:

The goal of this feature engineering project is to improve Delhivery's logistics operations by analyzing and transforming raw data into meaningful features that can be used to enhance forecasting models, optimize delivery routes, and increase overall operational efficiency. The project focuses on data cleaning, feature extraction, outlier detection, and transformations to ensure that the data is ready for advanced analytics and machine learning models.

Techniques Used:

Data Cleaning: Handling missing values and formatting datetime columns.

Feature Engineering: Creating new features like trip duration categories, time of day periods, and efficiency metrics (e.g., time efficiency, distance efficiency).

Skewness Treatment: Applying log or Box-Cox transformations to reduce skewness in data.
Aggregation: Grouping data by trip UUID and segment key to generate aggregate features like total time, distance, and averages.

Analysis Approach: Statistical and Visual Analysis: Identifying relationships between features using correlation heatmaps and handling outliers with boxplots.

Time-Based Insights: Creating features that categorize trips based on time of day and duration, helping improve forecasting accuracy.

Expected Outcome: The result will be a cleaned and feature-enriched dataset that can improve delivery time predictions, route optimization, and overall operational efficiency.

Expected Outcomes:

The ultimate goal of the project is to provide Delhivery with a cleaned, transformed, and feature-rich dataset that can be used for building robust predictive models. The feature engineering steps will enhance the company's ability to forecast delivery times, optimize delivery routes, and improve overall logistics efficiency.

By applying these techniques, Delhivery will be able to:

Optimize delivery time predictions by considering time-based features like trip duration categories and time of day.

Increase operational efficiency by analyzing route efficiency and identifying bottlenecks in the delivery process.

Enhance the quality of forecasting models by transforming skewed data and aggregating key features.

1.Importing and Understanding the data.

Code:

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# For handling datetime
from datetime import datetime

# For preprocessing
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
data = pd.read_csv('Downloads/delhivery_data.csv')
```

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	destination_center	destination_name	
0	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	0
1	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	0
2	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	0
3	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	0
4	training	2018-09-20 02:35:36.476840	thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...	Carting	trip-153741093647649320	IND388121AAA	Anand_VUNagar_DC (Gujarat)	IND388620AAB	Khambhat_MotvdDPP_D (Gujarat)	0

5 rows × 24 columns

2. Check for Null Values

Code:

```
data.isnull().sum()
```

Ouput:

```
data                            0
trip_creation_time              0
route_schedule_uuid             0
route_type                      0
trip_uuid                       0
source_center                   0
source_name                   293
destination_center              0
destination_name              261
od_start_time                   0
od_end_time                     0
start_scan_to_end_scan          0
is_cutoff                       0
cutoff_factor                   0
cutoff_timestamp                0
actual_distance_to_destination  0
actual_time                     0
osrm_time                       0
osrm_distance                   0
factor                          0
segment_actual_time             0
segment_osrm_time               0
segment_osrm_distance           0
segment_factor                  0
dtype: int64
```

3. Checking the datatypes/Info of data.

Code.

data.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144867 entries, 0 to 144866
Data columns (total 24 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   data                            144867 non-null  object
 1   trip_creation_time              144867 non-null  object
 2   route_schedule_uuid             144867 non-null  object
 3   route_type                      144867 non-null  object
 4   trip_uuid                       144867 non-null  object
 5   source_center                   144867 non-null  object
 6   source_name                     144574 non-null  object
 7   destination_center              144867 non-null  object
 8   destination_name                144606 non-null  object
 9   od_start_time                   144867 non-null  object
 10  od_end_time                     144867 non-null  object
 11  start_scan_to_end_scan          144867 non-null  float64
 12  is_cutoff                       144867 non-null  bool
 13  cutoff_factor                   144867 non-null  int64
 14  cutoff_timestamp                144867 non-null  object
 15  actual_distance_to_destination  144867 non-null  float64
 16  actual_time                     144867 non-null  float64
 17  osrm_time                       144867 non-null  float64
 18  osrm_distance                   144867 non-null  float64
 19  factor                          144867 non-null  float64
 20  segment_actual_time             144867 non-null  float64
 21  segment_osrm_time               144867 non-null  float64
 22  segment_osrm_distance           144867 non-null  float64
 23  segment_factor                  144867 non-null  float64
dtypes: bool(1), float64(10), int64(1), object(12)
memory usage: 25.6+ MB
```

4. Handling Missing Values and Formatting Data Types

In the dataset, the categorical columns ‘source_name’ and ‘destination_name’ contained missing values. To ensure consistency and avoid any impact on analysis, we handled these missing values by filling them with 'Unknown'. This approach preserves the data structure and ensures no rows are dropped unnecessarily.

Additionally, we formatted columns to their appropriate data types. Specifically, columns representing dates and times, such as ‘trip_creation_time’, ‘od_start_time’, ‘od_end_time’, and ‘cutoff_timestamp’, were converted to a datetime format to enable time-based analysis. These steps ensure the dataset is clean and ready for further processing.

Code:

```
# Filling null values with 'Unknown' for categorical columns
data['source_name'].fillna('Unknown', inplace=True)
data['destination_name'].fillna('Unknown', inplace=True)

#Converting columns to datetime format
datetime_columns = ['trip_creation_time', 'od_start_time', 'od_end_time', 'cutoff_timestamp']
for col in datetime_columns:
    data[col] = pd.to_datetime(data[col], errors='coerce')
```

Output:

```
Data types after conversion:
data                                    object
trip_creation_time                datetime64[ns]
route_schedule_uuid                     object
route_type                              object
trip_uuid                               object
source_center                           object
source_name                             object
destination_center                      object
destination_name                        object
od_start_time                     datetime64[ns]
od_end_time                       datetime64[ns]
start_scan_to_end_scan                   float64
is_cutoff                                   bool
cutoff_factor                              int64
cutoff_timestamp                  datetime64[ns]
actual_distance_to_destination           float64
actual_time                              float64
osrm_time                                float64
osrm_distance                            float64
factor                                   float64
segment_actual_time                      float64
segment_osrm_time                        float64
segment_osrm_distance                    float64
segment_factor                           float64
dtype: object
```

```
Null values after filling:
data                                 0
trip_creation_time                   0
route_schedule_uuid                  0
route_type                           0
trip_uuid                            0
source_center                        0
source_name                          0
destination_center                   0
destination_name                     0
od_start_time                        0
od_end_time                          0
start_scan_to_end_scan               0
is_cutoff                            0
cutoff_factor                        0
cutoff_timestamp                  3429
actual_distance_to_destination       0
actual_time                          0
osrm_time                            0
osrm_distance                        0
factor                               0
segment_actual_time                  0
segment_osrm_time                    0
segment_osrm_distance                0
segment_factor                       0
dtype: int64
```

Insights:

When converting columns like 'cutoff_timestamp' to the datetime format using pd.to_datetime, any values that are invalid or improperly formatted are automatically set to NaT (Not a Time), which is considered a null value in pandas. This behavior ensures that non-convertible values do not cause errors but are instead flagged as missing.

To address the null values in 'cutoff_timestamp' caused by the conversion, I took the following steps:

Flagging Missing Values: Created a new column, is_cutoff_missing, to indicate whether cutoff_timestamp was missing. This helps retain the information about rows with invalid or missing data.

Code:

```
#Flagging rows where cutoff_timestamp is missing
data['is_cutoff_missing'] = data['cutoff_timestamp'].isnull().astype(int)
```

Imputing Null Values: Replaced null values in cutoff_timestamp with a placeholder timestamp (1970-01-01 00:00:00). This ensures the column remains valid for further analysis while marking missing data explicitly.

Code:

```
# Filling missing cutoff_timestamp values with a placeholder
data['cutoff_timestamp'].fillna(pd.Timestamp('1970-01-01 00:00:00'), inplace=True)
```

5.Lets check for cutoff timestamp null values.

```
print(data['cutoff_timestamp'].isnull().sum())
```

Output:

```
: print(data['cutoff_timestamp'].isnull().sum())
0
```

Now the data is finally ready for further Analysis.

6. Creating Segment-Level Aggregates.

Code:

```
Create a segment key for grouping
data['segment_key'] = data['trip_uuid'] + "_" + data['source_center'] + "_" + data['destination_center']

# Aggregating segment_actual_time, segment_osrm_distance, and segment_osrm_time
data['segment_actual_time_sum'] = data.groupby('segment_key')['segment_actual_time'].cumsum()
data['segment_osrm_distance_sum'] = data.groupby('segment_key')['segment_osrm_distance'].cumsum()
data['segment_osrm_time_sum'] = data.groupby('segment_key')['segment_osrm_time'].cumsum()

# Display updated data
print("Data after creating segment-level aggregates:")
print(data.head())
```

In this step, I generated a **segment_key** to uniquely identify each segment of a trip. This key combines the trip_uuid, source_center, and destination_center columns, ensuring that segments can be grouped and analyzed individually.

Using this segment_key, we calculated cumulative sums for important features such as:

- segment_actual_time → **segment_actual_time_sum**: Represents the cumulative time taken for all segments of the trip up to the current segment.
- segment_osrm_distance → **segment_osrm_distance_sum**: Represents the cumulative distance covered for all segments up to the current segment.
- segment_osrm_time → **segment_osrm_time_sum**: Represents the cumulative OSRM-estimated time for all segments up to the current segment.

This aggregation helps us understand the overall performance of a trip by summing up segment-level metrics while preserving the segment-specific details for analysis.

Output:

```
Data after creating segment-level aggregates:
       data         trip_creation_time  \
0  training 2018-09-20 02:35:36.476840
1  training 2018-09-20 02:35:36.476840
2  training 2018-09-20 02:35:36.476840
3  training 2018-09-20 02:35:36.476840
4  training 2018-09-20 02:35:36.476840

                              route_schedule_uuid route_type  \
0  thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...    Carting
1  thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...    Carting
2  thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...    Carting
3  thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...    Carting
4  thanos::sroute:eb7bfc78-b351-4c0e-a951-fa3d5c3...    Carting

                trip_uuid source_center                source_name  \
0  trip-153741093647649320  IND388121AAA  Anand_VUNagar_DC (Gujarat)
1  trip-153741093647649320  IND388121AAA  Anand_VUNagar_DC (Gujarat)
2  trip-153741093647649320  IND388121AAA  Anand_VUNagar_DC (Gujarat)
3  trip-153741093647649320  IND388121AAA  Anand_VUNagar_DC (Gujarat)
4  trip-153741093647649320  IND388121AAA  Anand_VUNagar_DC (Gujarat)

  destination_center              destination_name  \
0       IND388620AAB  Khambhat_MotvdDPP_D (Gujarat)
1       IND388620AAB  Khambhat_MotvdDPP_D (Gujarat)
2       IND388620AAB  Khambhat_MotvdDPP_D (Gujarat)
3       IND388620AAB  Khambhat_MotvdDPP_D (Gujarat)
4       IND388620AAB  Khambhat_MotvdDPP_D (Gujarat)


              od_start_time  ...    factor  segment_actual_time  \
 0 2018-09-20 03:21:32.418600  ...  1.272727                14.0
 1 2018-09-20 03:21:32.418600  ...  1.200000                10.0
 2 2018-09-20 03:21:32.418600  ...  1.428571                16.0
 3 2018-09-20 03:21:32.418600  ...  1.550000                21.0
 4 2018-09-20 03:21:32.418600  ...  1.545455                 6.0

    segment_osrm_time  segment_osrm_distance segment_factor  is_cutoff_missing  \
 0               11.0                11.9653       1.272727                  0
 1                9.0                 9.7590       1.111111                  0
 2                7.0                10.8152       2.285714                  1
 3               12.0                13.0224       1.750000                  0
 4                5.0                 3.9153       1.200000                  0

                                     segment_key  segment_actual_time_sum  \
 0  trip-153741093647649320_IND388121AAA_IND388620AAB                     14.0
 1  trip-153741093647649320_IND388121AAA_IND388620AAB                     24.0
 2  trip-153741093647649320_IND388121AAA_IND388620AAB                     40.0
 3  trip-153741093647649320_IND388121AAA_IND388620AAB                     61.0
 4  trip-153741093647649320_IND388121AAA_IND388620AAB                     67.0

    segment_osrm_distance_sum  segment_osrm_time_sum
 0                    11.9653                   11.0
 1                    21.7243                   20.0
 2                    32.5395                   27.0
 3                    45.5619                   39.0
 4                    49.4772                   44.0

 [5 rows x 29 columns]
```

Insights from Segment-Level Aggregates

The segment-level aggregation provides valuable insights into the cumulative performance of trips by summing up segment-specific metrics. Here are some key observations based on the displayed data:

---

1. Cumulative Time Analysis (segment_actual_time_sum)

- The column **segment_actual_time_sum** reflects the cumulative actual time taken for all segments of a trip.
- For example:
  - For segment_key = trip-153741093647649320_IND388121AAA_IND388620AAB, the cumulative actual time increases as more segments are added:
    - First segment: 14.0 minutes
    - Second segment: 24.0 minutes (cumulative)
    - By the 5th segment: 67.0 minutes (cumulative)
- **Insight**: This metric helps track how total trip times accumulate over multiple segments, enabling the identification of delays across specific segments.

---

2. Cumulative Distance Analysis (segment_osrm_distance_sum)

- The column **segment_osrm_distance_sum** shows the cumulative distance (in km) computed by OSRM for all segments.
- For the same segment_key:
  - First segment: 11.9653 km
  - Second segment: 21.7243 km
  - By the 5th segment: 49.4772 km
- **Insight**: This metric is useful for evaluating whether the actual distance aligns with expected cumulative distances for long-haul routes. Any significant deviations might indicate routing inefficiencies.

---

3. Cumulative OSRM Time Analysis (segment_osrm_time_sum)

- The column **segment_osrm_time_sum** aggregates the OSRM-estimated time for all segments.
- For the same segment_key:
  - First segment: 11.0 minutes
  - Second segment: 20.0 minutes
  - By the 5th segment: 44.0 minutes
- **Insight**: Comparing **segment_actual_time_sum** and **segment_osrm_time_sum** reveals route efficiency. For instance:
  - Actual cumulative time (67.0 minutes) exceeds OSRM-estimated time (44.0 minutes), indicating potential delays or inefficiencies.

---

4. Segment Key as a Unique Identifier

- The **segment_key** uniquely identifies segments within a trip, combining trip_uuid, source_center, and destination_center.
- **Insight**: This grouping allows a granular view of trips, enabling analysis of performance at both segment and trip levels.

---

5. Potential Bottlenecks and Delays

- By tracking cumulative metrics, segments with significant deviations in time or distance can be flagged for further investigation.
- **Insight**: Delays may arise from specific route inefficiencies or operational challenges (e.g., traffic, waiting times at hubs).

7. Segment-Level Aggregation.
Code:

```
# Create a dictionary for aggregation
create_segment_dict = {
   'trip_uuid': 'first',  # Keep first unique value
   'source_center': 'first',
   'destination_center': 'first',
   'segment_actual_time_sum': 'max',  # Max cumulative value
   'segment_osrm_distance_sum': 'max',
   'segment_osrm_time_sum': 'max',
   'route_type': 'first',  # Keeping first value for categorical data
}

# Perform aggregation
segment_data = data.groupby('segment_key').agg(create_segment_dict).reset_index()

# Display segment-level aggregated data
print("Segment-level aggregated data:")
print(segment_data.head())
```

Using the create_segment_dict dictionary, we performed aggregation on the segment_key groups to summarize key metrics for each unique segment of a trip. This approach consolidates segment-level data into a concise and meaningful format by retaining only the most relevant values for analysis.

Logic:

1. **Grouping by segment_key**:
   - Each segment_key uniquely identifies a segment based on the combination of trip_uuid, source_center, and destination_center.
2. **Aggregation Logic**:
   - **Categorical Columns**:
     - For columns like trip_uuid, source_center, destination_center, and route_type, we retained the **first occurrence** as these values remain consistent within a segment.
   - **Numerical Columns**:
     - For cumulative metrics (segment_actual_time_sum, segment_osrm_distance_sum, segment_osrm_time_sum), we used the **maximum value** to represent the final cumulative figure for the segment.
3. **Resetting the Index**:
   - After grouping and aggregating, we reset the index to return the results in a DataFrame format for further analysis.

Why This Approach?

1. **Summarization**:
   - Aggregating data ensures that each segment's performance is represented by key summary metrics, reducing redundancy while retaining meaningful insights.
2. **Actionable Metrics**:
   - The maximum cumulative values for time and distance provide the total performance metrics for each segment, which can be directly compared across trips or routes.
3. **Preparation for Analysis**:
   - The aggregated dataset is now cleaner and more concise, making it easier to analyze segment-level trends and identify inefficiencies.

Insights from Aggregated Data:

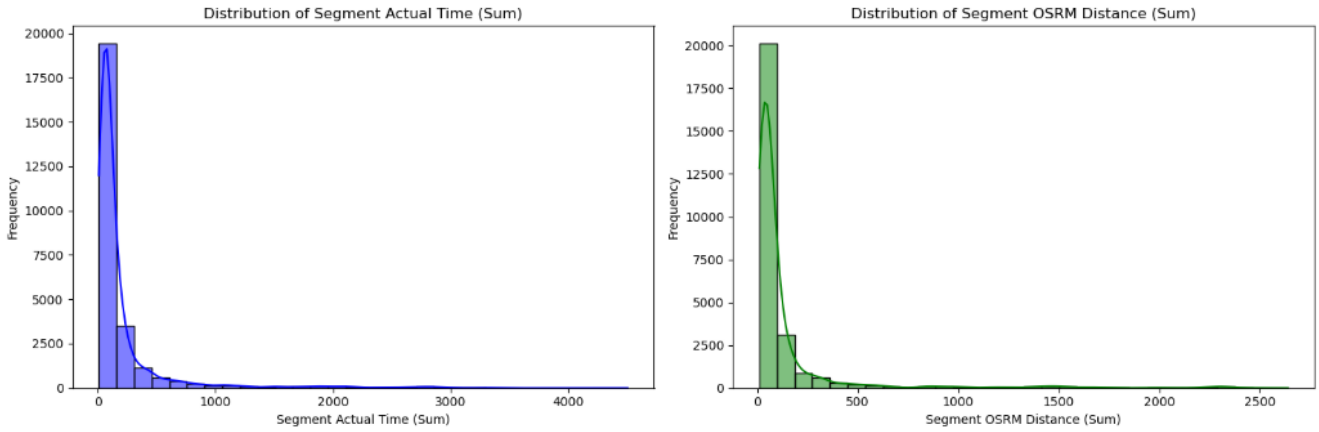
1. **Segment Performance**:
   - By analyzing the segment_actual_time_sum and segment_osrm_time_sum, we can assess whether segments are efficient or experiencing delays.
2. **Route Type Trends**:
   - Using the route_type column, we can compare the performance of different transportation methods (e.g., "Carting" vs. "FTL").
3. **Distance Consistency**:
   - Comparing segment_osrm_distance_sum across segments helps evaluate whether the OSRM-estimated distances align with actual distances.

8. Let's Visualize the distribution of cumulative time and distance metrics.

Code:

```
# Visualize the distribution of aggregated time and distance metrics
plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
sns.histplot(segment_data['segment_actual_time_sum'], kde=True, bins=30, color='blue')
plt.title("Distribution of Segment Actual Time (Sum)")
plt.xlabel("Segment Actual Time (Sum)")
plt.ylabel("Frequency")

plt.subplot(1, 2, 2)
sns.histplot(segment_data['segment_osrm_distance_sum'], kde=True, bins=30, color='green')
plt.title("Distribution of Segment OSRM Distance (Sum)")
plt.xlabel("Segment OSRM Distance (Sum)")
plt.ylabel("Frequency")

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

Output:



Insights:

Skewed Distribution

- Both Segment Actual Time (Sum) and Segment OSRM Distance (Sum) have a **highly right-skewed distribution**.
- This indicates that the majority of segments have small cumulative times and distances, but there are a few segments with significantly higher values (outliers).

Most Common Values

- **For Segment Actual Time (Sum)**:
  - Most values are concentrated near zero (likely below 500).
  - This suggests that for most segments, actual travel times are relatively short.
- **For Segment OSRM Distance (Sum)**:
  - Similarly, most distances are short, concentrated below 200.
  - This implies that the majority of trips involve short travel distances.

9. Performing feature engineering to enhance the dataset with additional insights and structured information for further analysis.

Code:

data['od_time_diff_hour'] = (data['od_end_time'] - data['od_start_time']).dt.total_seconds() / 3600

Purpose: This computes the time difference (in hours) between the od_start_time (start of the trip) and od_end_time (end of the trip).

Result: A new column, od_time_diff_hour, that indicates the trip duration in hours.

Insights:

- This can be used to analyze the duration of trips and identify:
  - Average trip durations.
  - Long-duration trips (possible inefficiencies).
  - Correlations between trip durations and other features (e.g., distance or region).

Extract Features from trip_creation_time:
Code:
```
data['trip_year'] = data['trip_creation_time'].dt.year
data['trip_month'] = data['trip_creation_time'].dt.month
data['trip_day'] = data['trip_creation_time'].dt.day
data['trip_weekday'] = data['trip_creation_time'].dt.weekday
```

Insights:

- These features allow for **time-based analysis**, such as:
  - Identifying trends or seasonality in trips (e.g., higher demand in certain months or days of the week).
  - Comparing trip behavior across different time periods (e.g., weekdays vs weekends).
  - Exploring if trip duration or distance varies by time of year.

Split Source and Destination Names into City and State:
Code:
```
data[['source_city', 'source_state']] = data['source_name'].str.extract(r'(.+?)\s\((.+?)\)')
data[['destination_city', 'destination_state']] = data['destination_name'].str.extract(r'(.+?)\s\((.+?)\)')
```

- **Purpose**: Extracts city and state information from source_name and destination_name columns (which might have combined information like City (State)).
- **Result**: Creates source_city, source_state, destination_city, and destination_state.

Insights:

- Enables **geographical analysis**, such as:
  - Trip patterns between specific cities or states.
  - Comparing trip behavior across different regions.
  - Analyzing if certain states or cities contribute more to high trip durations or inefficiencies.

Drop Original Columns (source_name and destination_name):
Code: `data.drop(['source_name', 'destination_name'], axis=1, inplace=True)`

Purpose: Removes columns no longer needed after splitting the information into separate fields.

10. Duration-Based Analysis:

- **Objective**: Compare trip durations across time, cities, or states, and identify outliers (i.e., trips with very high durations).

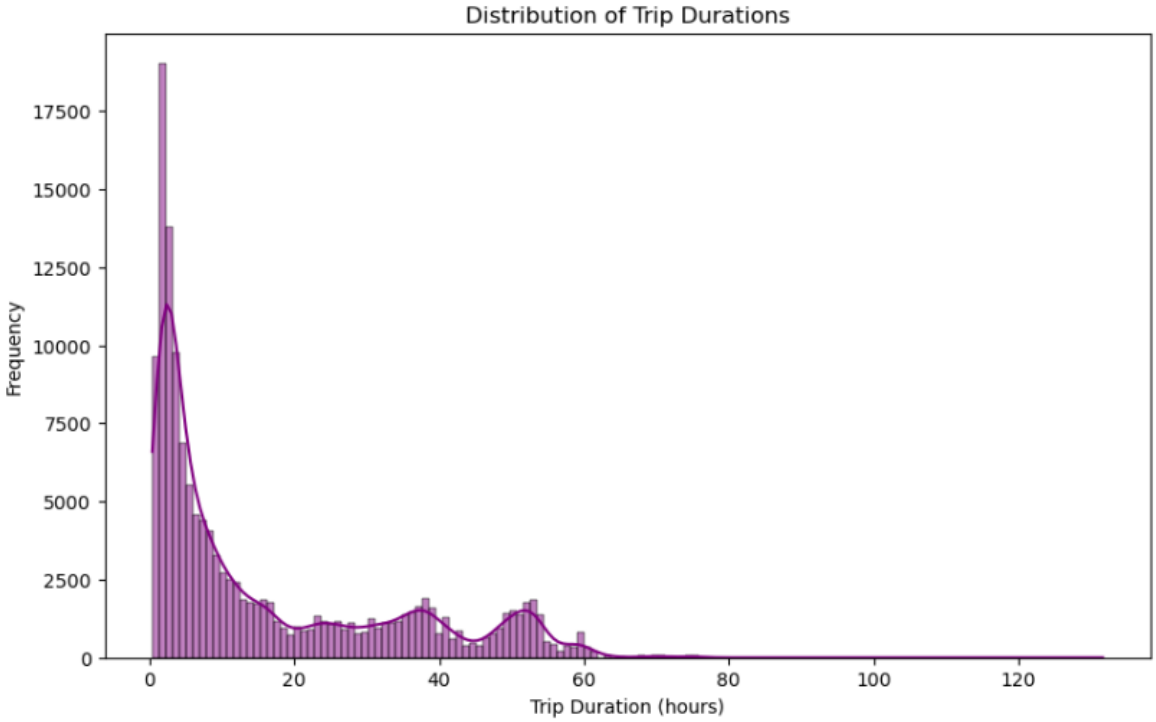
a. Trip Duration Distribution:

This will help to check if there are any outliers in the trip durations.

Code:

```
plt.figure(figsize=(10, 6))
sns.histplot(data['od_time_diff_hour'], kde=True, color='purple')
plt.title('Distribution of Trip Durations')
plt.xlabel('Trip Duration (hours)')
plt.ylabel('Frequency')
plt.show()
```

Output:



Insights:

Overall Distribution:

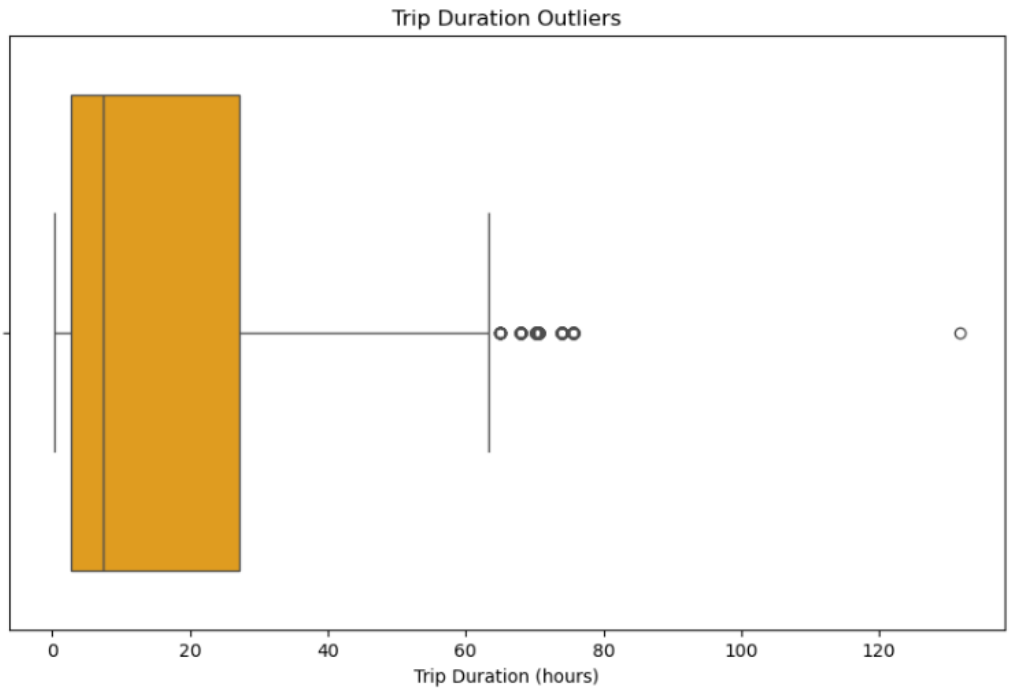
The trip durations exhibit a right-skewed distribution, indicating that a majority of trips are shorter in duration.
There is a significant peak in the distribution around 0-5 hours, suggesting a large number of short trips.
The distribution tails off gradually, implying a smaller number of longer trips.

b. Identifying Outliers:

We can use a box plot to visually detect outliers in trip durations.

Code:

```
plt.figure(figsize=(10, 6))
sns.boxplot(x=data['od_time_diff_hour'], color='orange')
plt.title('Trip Duration Outliers')
plt.xlabel('Trip Duration (hours)')
plt.show()
```



Insights:

Typical Trip Duration:

- The interquartile range (IQR), which spans from Q1 (25th percentile) to Q3 (75th percentile), indicates that most trip durations fall between approximately 0 to 20 hours.
- The median trip duration is closer to the lower end of this range, suggesting that shorter trips are more common.

Outliers:

- The data contains several outliers, with trip durations exceeding 40 hours.
- One extreme outlier is observed beyond 120 hours, which significantly deviates from the rest of the dataset.

Skewness:

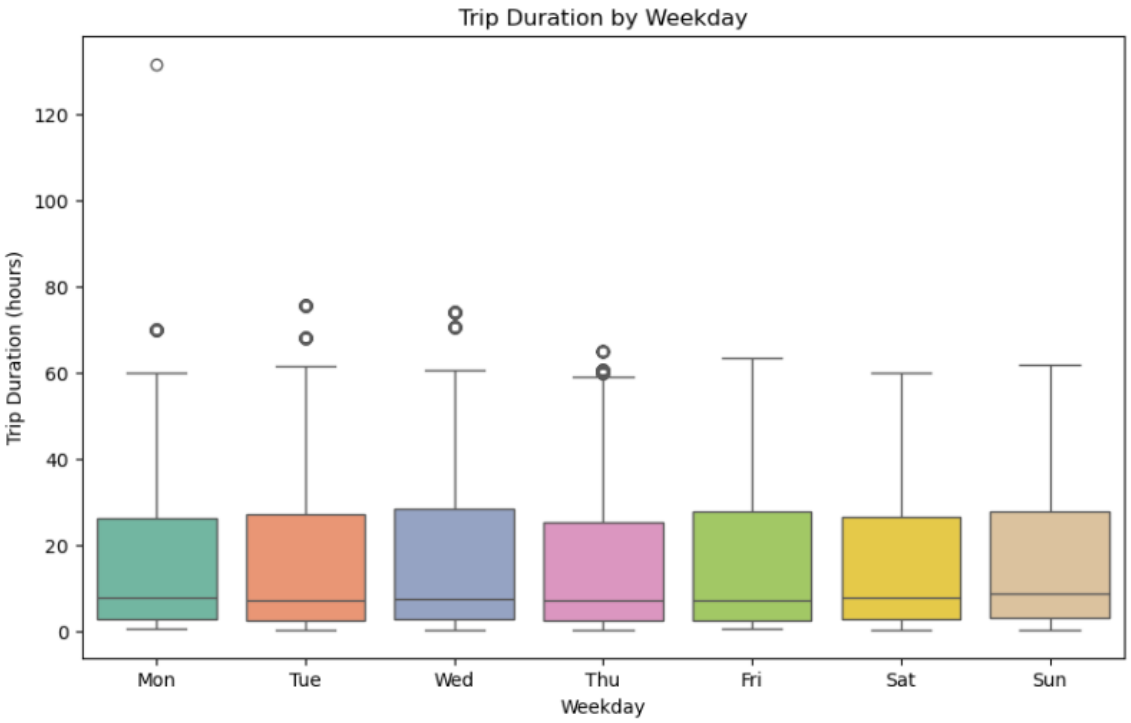
- The distribution is right-skewed, as indicated by the presence of long-duration outliers. This suggests that while most trips are completed within a reasonable timeframe, there are a few instances of unusually long trips.

11. Time-Based Trends:

Objective: Analyze the trip data based on time (e.g., busiest months, weekdays) and assess the variation of trip characteristics (e.g., duration, frequency) during weekends vs weekdays.

```
plt.figure(figsize=(10, 6))
sns.countplot(x=data['trip_month'], palette='coolwarm')
plt.title('Number of Trips by Month')
plt.xlabel('Month')
plt.ylabel('Number of Trips')
plt.xticks(ticks=range(12), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.show()
```

Output:



Median Trip Durations: The median trip durations appear relatively consistent across the weekdays, falling within a similar range, indicating no significant differences in the typical delivery time based on the day of the week.

Interquartile Range (IQR): The IQR (the box portion) is also similar for most weekdays, suggesting a consistent variation in trip durations irrespective of the day.

Outliers: Outliers are observed on all weekdays, with the most extreme outlier (greater than 120 hours) on Monday.

Other notable outliers (40–80+ hours) are present on Tuesday, Wednesday, and Thursday, indicating potential operational delays or data recording issues on these days.

Day-Specific Observations: Thursday shows slightly more variability compared to other days, as its whiskers are longer and encompass a wider range of durations.

Weekends (Saturday and Sunday) seem to have fewer extreme outliers, possibly reflecting reduced operational load or more streamlined deliveries during these days.

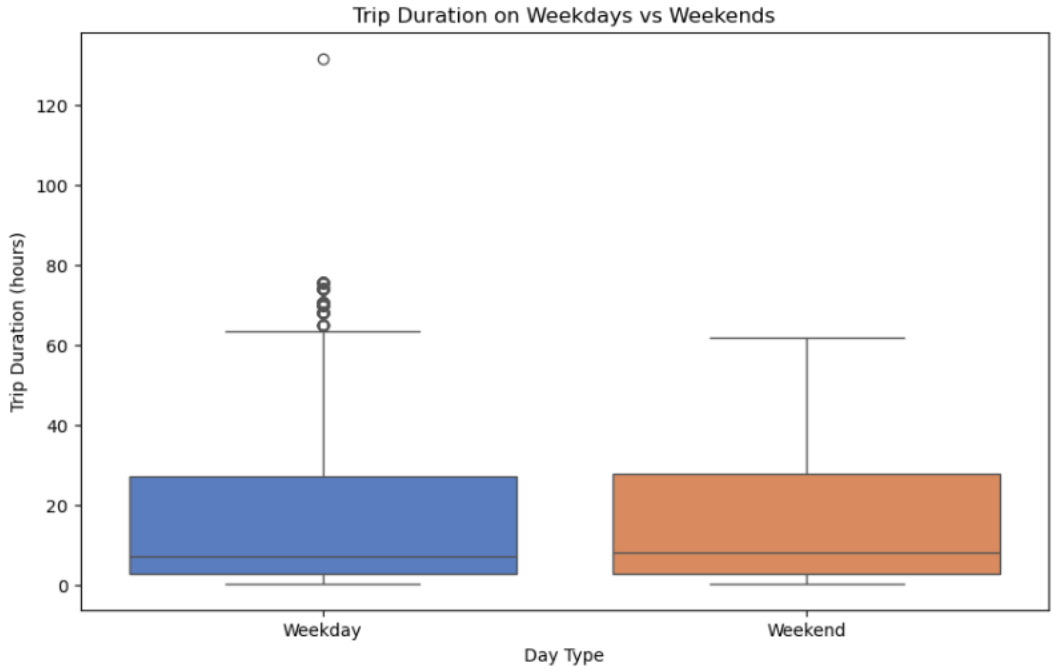
12. Weekend vs Weekday Comparison:

Compares trip durations on weekdays vs weekends.

Code:

```
data['weekend'] = data['trip_weekday'].apply(lambda x: 'Weekend' if x >= 5 else 'Weekday')

plt.figure(figsize=(10, 6))
sns.boxplot(x=data['weekend'], y=data['od_time_diff_hour'], palette='muted')
plt.title('Trip Duration on Weekdays vs Weekends')
plt.xlabel('Day Type')
plt.ylabel('Trip Duration (hours)')
plt.show()
```

Output:



Insights:

Median Trip Duration: The median trip duration is slightly higher for weekdays compared to weekends. This suggests that weekday trips might face more challenges such as traffic congestion or higher delivery volumes.

Interquartile Range (IQR): Weekdays have a larger IQR (box width) than weekends, indicating greater variability in trip durations during weekdays.

The narrower IQR for weekends suggests that trip durations are more consistent on Saturdays and Sundays.

Outliers:

Weekdays: Several outliers are visible, with one extreme outlier exceeding 120 hours, which might indicate operational delays or errors in data logging.

Weekends: No extreme outliers are present, highlighting smoother operations or fewer disruptions during the weekends.

Whiskers (Range): Weekday trips exhibit a wider range compared to weekends, with the upper whisker extending beyond 60 hours. This implies that longer trips are more common during weekdays.

Potential Causes for Weekday Variability: High traffic density and increased delivery demand during weekdays could explain the greater variability and outliers.

Weekends may involve fewer deliveries and optimized logistics, leading to more predictable trip durations.

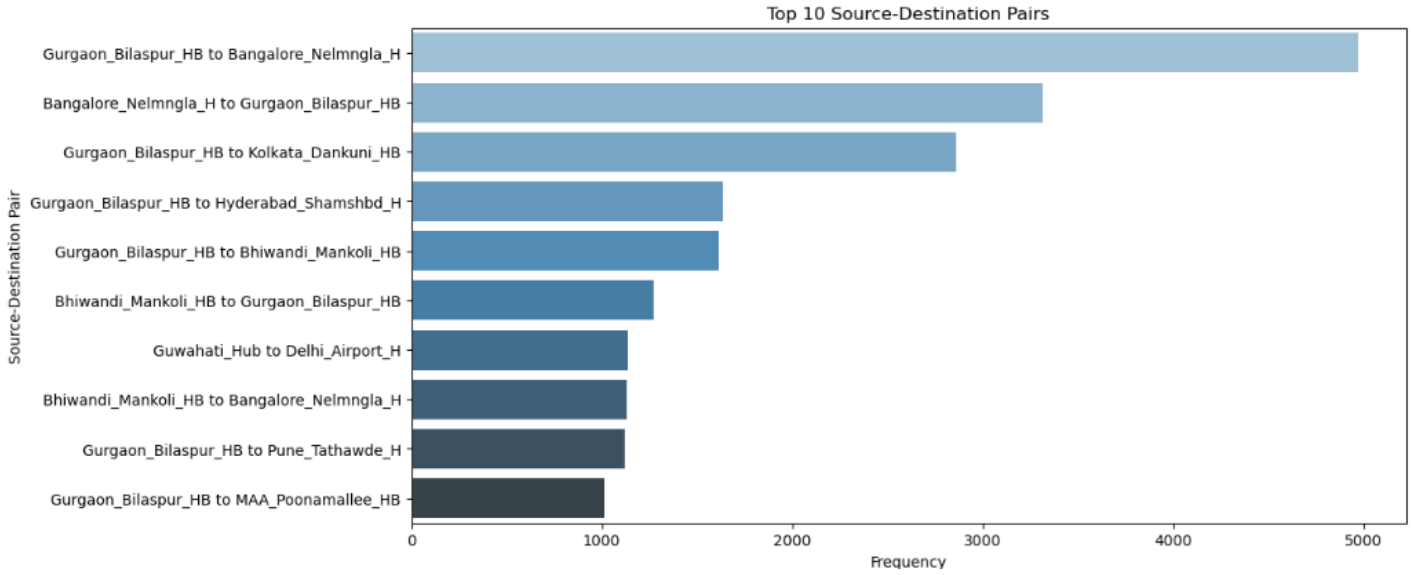
Geographical Patterns:

Objective: Examine the relationships between source and destination locations (cities or states), and analyze how trip durations vary across these locations.

13. Top Source-Destination Pairs: Visualize the most frequent source-destination pairs.

Code:

```
# Create a new column to combine source and destination
data['source_destination'] = data['source_city'] + ' to ' + data['destination_city']

# Plot top 10 most frequent source-destination pairs
top_pairs = data['source_destination'].value_counts().head(10)

plt.figure(figsize=(12, 6))
sns.barplot(x=top_pairs.values, y=top_pairs.index, palette='Blues_d')
plt.title('Top 10 Source-Destination Pairs')
plt.xlabel('Frequency')
plt.ylabel('Source-Destination Pair')
plt.show()
```



Insights:

- **Gurgaon-Bilaspur dominance:** This route is the most frequent, appearing in multiple top pairs.
- **Bangalore-Nelmangla popularity:** This is a popular destination point.
- **Reciprocal routes:** Some routes have significant traffic in both directions.
- **Regional focus:** The top pairs seem to be within India, with specific hubs and regions being prominent.

By understanding these patterns can help optimize operations, allocate resources, and tailor marketing strategies.

14. Aggregate data at the trip level

Objective

The goal of this code is to aggregate data at the **trip level**. This means consolidating information about individual trip segments into a single summary record for each unique trip (identified by trip_uuid). Aggregating at the trip level provides a higher-level view of trips and helps in analyzing overall trip patterns and statistics.

Code:

```
# Dictionary for trip-level aggregation
create_trip_dict = {
    'segment_actual_time_sum': 'sum',  # Sum over all segments
    'segment_osrm_distance_sum': 'sum',
    'segment_osrm_time_sum': 'sum',
    'od_time_diff_hour': 'mean',  # Average time difference
    'trip_year': 'first',  # Keep first value for categorical data
    'trip_month': 'first',
}

# Aggregate at trip level
trip_data = data.groupby('trip_uuid').agg(create_trip_dict).reset_index()

# Display trip-level aggregated data
print("Trip-level aggregated data:")

print(trip_data.head())
```

Output:

```
Trip-level aggregated data:
                 trip_uuid  segment_actual_time_sum  \
0  trip-153671041653548748                  15516.0
1  trip-153671042288605164                    396.0
2  trip-153671043369099517                 110876.0
3  trip-153671046011330457                     82.0
4  trip-153671052974046625                    555.0

   segment_osrm_distance_sum  segment_osrm_time_sum  od_time_diff_hour  \
0                  14222.1104                10670.0          18.666877
1                    268.5094                  199.0           1.691063
2                  90479.7151                70971.0          46.571982
3                     31.8441                   25.0           1.674916
4                    266.2915                  204.0           3.782645

   trip_year  trip_month
0       2018           9
1       2018           9
2       2018           9
3       2018           9
4       2018           9
```

Insights:

Aggregation at Trip Level: Each trip (trip_uuid) is now represented by a single row, with aggregated information about the trip's duration, distance, time difference, and categorical attributes.

Simplified Data for Analysis: The data is now summarized at a higher level, reducing redundancy and complexity. This makes it easier to perform further analysis.

Conclusions:

1. **Trip Duration (segment_actual_time_sum)**:
   - Trips show a large variation in duration, from short (~6 minutes) to very long (~30 hours). This indicates diverse trip types and possible operational delays for longer trips.
2. **Trip Distance (segment_osrm_distance_sum)**:
   - Distances range from small (31.84 units) to extensive (90,479 units), suggesting the dataset covers local deliveries to long-haul trips.
3. **Trip Efficiency**:
   - Significant discrepancies between actual time and OSRM time for some trips (e.g., trip-153671043369099517) indicate potential delays due to traffic, routing, or logistical issues.
4. **Segment Durations (od_time_diff_hour)**:
   - Average segment durations vary widely, with some indicating potential inefficiencies in completing segments.
5. **Time Attributes**:
   - All trips in this sample are from September 2018, so broader temporal trends can't be identified without additional data.

15. Outlier Detection Using IQR:

Objective: Identify and handle outliers in key numerical features to avoid their impact on downstream analyses.

Code:

```
# Outlier detection using IQR
for col in ['segment_actual_time_sum', 'segment_osrm_distance_sum', 'od_time_diff_hour']:
    Q1 = data[col].quantile(0.25)
    Q3 = data[col].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    data[col] = np.clip(data[col], lower, upper)  # Clip outliers

# Normalize numerical features
scaler = MinMaxScaler()
numerical_cols = ['segment_actual_time_sum', 'segment_osrm_distance_sum', 'od_time_diff_hour']
data[numerical_cols] = scaler.fit_transform(data[numerical_cols])

# Display normalized data
print("Data after normalization:")
print(data.head())
```

Output:

The cleaned and normalized dataset retains meaningful information while mitigating the effects of outliers and scale differences.

16. Hypothesis Testing:

Objective of the Test

The goal of this hypothesis test was to determine whether there is a statistically significant difference between **actual_time** (the observed trip durations) and **osrm_time** (the predicted trip durations by OSRM). This analysis aims to assess whether OSRM's predictions align with reality or if there are systematic inaccuracies in its predictions.

Approach:

1. **Paired T-Test**:
    - Since the same trips have both actual_time and osrm_time, the test compares paired observations to see if the mean difference is significantly different from zero.
    - The **null hypothesis (H0)**: The mean difference between actual_time and osrm_time is zero
    - The **alternative hypothesis (H1)**: The mean difference is not zero

Bootstrap Sampling: To validate the robustness of the t-test results, we used bootstrap sampling to estimate the mean difference across 1,000 resamples. This approach doesn't assume normality and provides additional confidence in the results.

Mean Difference: The mean difference was calculated to quantify the average discrepancy between actual_time and osrm_time.

Code:

```
sample = data.sample(n=1000, random_state=42)

# Paired t-test on sampled data
stat, p_value = ttest_rel(sample['actual_time'], sample['osrm_time'])
print(f"Paired T-Test on Sample: Statistic={stat}, P-value={p_value}")
```

Output:

```
Paired T-Test on Sample: Statistic=21.528082381891217, P-value=9.427738103753322e-85
```

Bootstrap Sampling:

```
bootstrap_stats = []
for _ in range(1000):  # Generate 1000 bootstrap samples
    sample = data.sample(n=1000, random_state=None)
    stat, _ = ttest_rel(sample['actual_time'], sample['osrm_time'])
    bootstrap_stats.append(stat)

print(f"Bootstrap Mean Statistic: {np.mean(bootstrap_stats)}")
```

```
Bootstrap Mean Statistic: 21.17234947908243
```

Results:

~ **Paired T-Test**:

- **Test Statistic**: 21.5321.5321.53
- **P-value**: 9.43×10^{-85} (essentially zero)
- **Interpretation**: The p-value is far below the standard significance threshold ($\alpha=0.05$), leading to rejection of the null hypothesis. This indicates a statistically significant difference between actual_time and osrm_time.

~**Bootstrap Sampling**:

- **Bootstrap Mean Statistic**: 21.17
- **Interpretation**: The bootstrap results confirmed the t-test's findings, showing consistent test statistics across multiple samples. This validates the robustness of the observed differences.

~**Mean Difference**:

- **Value**: 205.09 seconds (approximately 3.42minutes).
- **Interpretation**: On average, the actual_time exceeds osrm_time by 205.09 seconds. This suggests that OSRM systematically underestimates trip durations, which could have operational implications.

Insights:

Statistical Conclusion:

- The t-test results show a highly significant difference between actual_time and osrm_time.
- The large test statistic and low p-value indicate strong evidence that OSRM predictions differ from actual durations.

Practical Significance:

- The mean difference of 205.09seconds is meaningful in a practical context:
  - For short trips, a 3-minute underestimation could lead to operational inefficiencies or customer dissatisfaction.
  - For longer trips, this consistent underestimation might compound and affect scheduling or resource allocation.

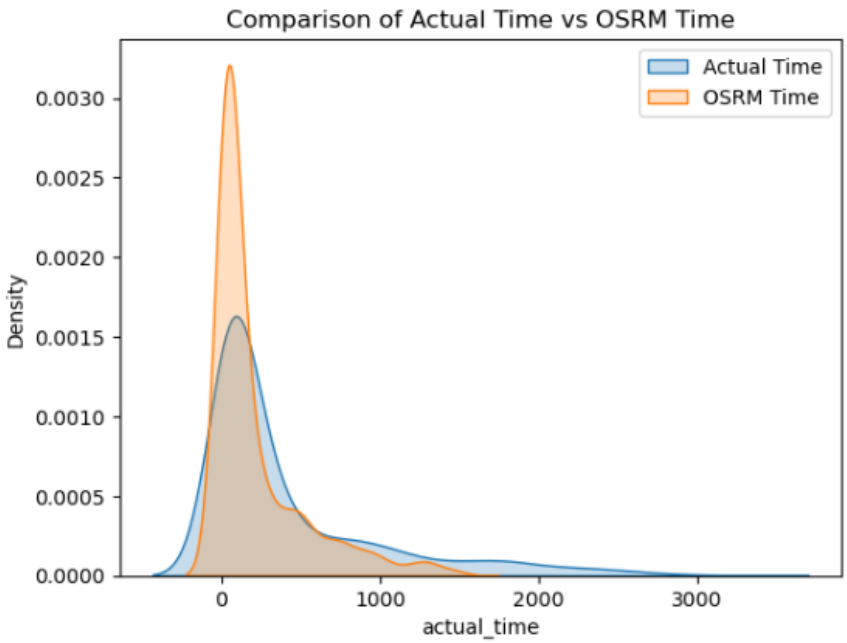
Summary

The hypothesis test revealed a statistically and practically significant difference between actual and predicted trip durations. The mean difference of 205.09 seconds highlights an underestimation in OSRM predictions, providing actionable insights for improving accuracy and operational efficiency.

17. Visualization:
Distribution of actual_time and osrm_time to visually assess the difference.
Code:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.kdeplot(sample['actual_time'], label='Actual Time', shade=True)
sns.kdeplot(sample['osrm_time'], label='OSRM Time', shade=True)
plt.legend()
plt.title("Comparison of Actual Time vs OSRM Time")
plt.show()
```

Output:



Insights:

Skewness: Both the actual and OSRM times exhibit a right-skewed distribution. This indicates that a majority of the travel times are concentrated towards the lower end, with a few outliers extending to significantly higher values.

Difference in Central Tendency: While both distributions have a similar shape, the OSRM times appear to be slightly shifted to the right compared to the actual times. This suggests that, on average, OSRM tends to overestimate travel times.

Variability: The actual travel times seem to have a slightly wider spread than the OSRM times. This indicates that there is more variability in actual travel times, likely due to factors like traffic conditions, road closures, and unforeseen events.

Recommendations:

Calibrate OSRM Model: To improve accuracy, the OSRM model could be calibrated using historical traffic data and real-time traffic feeds. This could help in better accounting for factors like congestion and road incidents.

Provide Uncertainty Ranges: Instead of providing a single estimated time, OSRM could provide a range of possible travel times, reflecting the uncertainty inherent in real-world conditions.

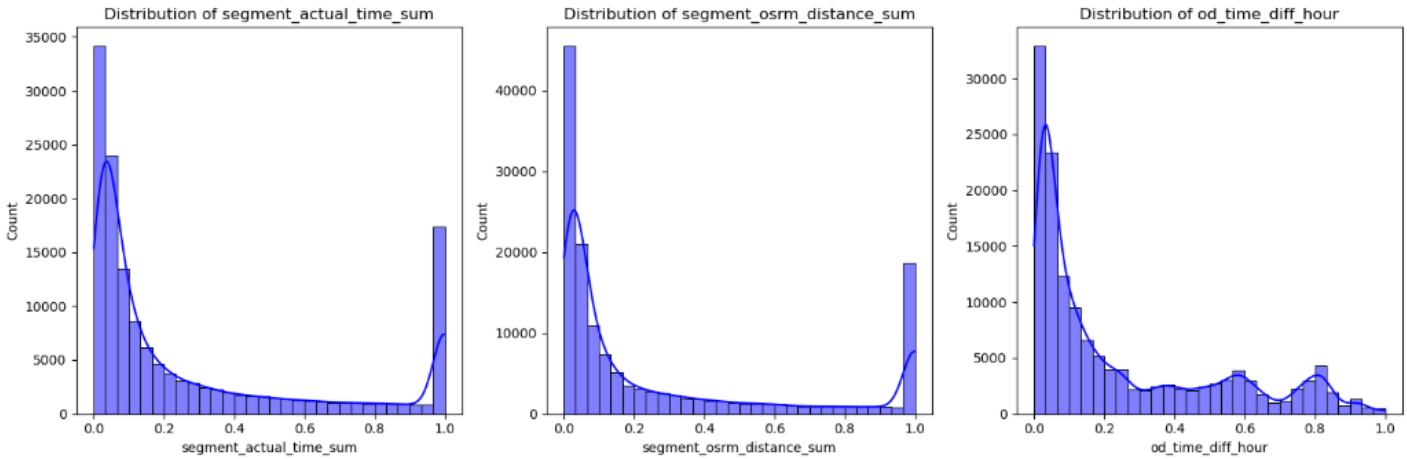
Consider User Feedback: Incorporating user-reported travel times can further refine the model and improve its accuracy over time.

18. Distribution plots for numerical features

Code:

```
numerical_features = ['segment_actual_time_sum', 'segment_osrm_distance_sum', 'od_time_diff_hour']

plt.figure(figsize=(15, 5))
for i, feature in enumerate(numerical_features, 1):
    plt.subplot(1, 3, i)
    sns.histplot(data[feature], kde=True, bins=30, color='blue')
    plt.title(f"Distribution of {feature}")
plt.tight_layout()
plt.show()
```

Output:



Insights:

Distribution of segment_actual_time_sum:

Skewness: The distribution is heavily right-skewed, indicating that most of the actual travel times are concentrated towards the lower end, with a few outliers extending to significantly higher values. This suggests that there are instances of unusually long travel times.

Distribution of segment_osrm_distance_sum:

Skewness: Similar to the actual travel times, the OSRM distances also exhibit a right-skewed distribution. This indicates that OSRM tends to overestimate distances in some cases.

Distribution of od_time_diff_hour:

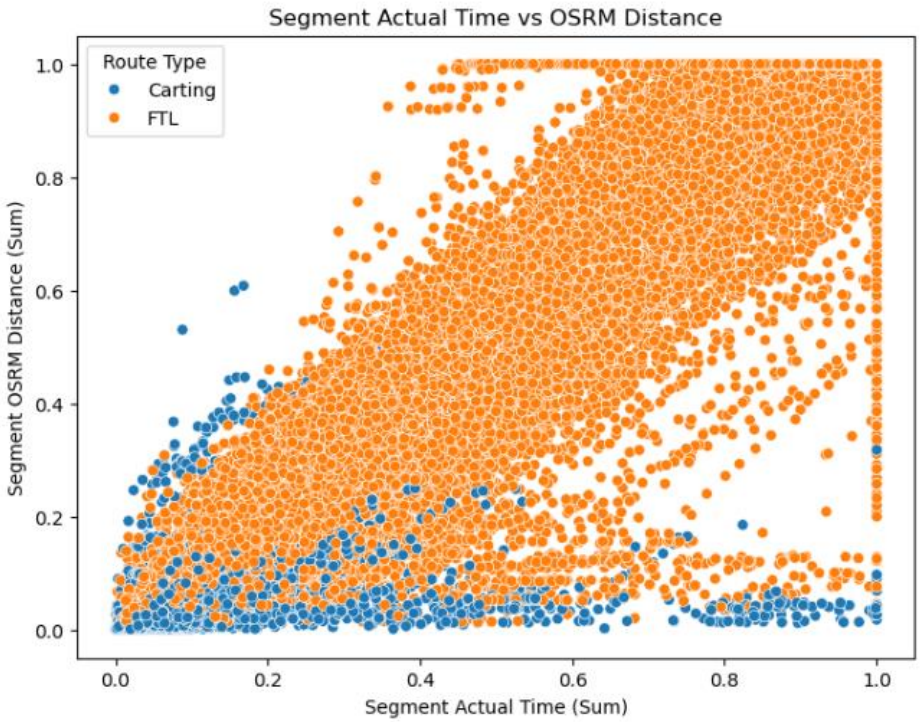
Central Tendency: The distribution is centered around 0, suggesting that, on average, OSRM's estimated travel times are not significantly different from the actual travel times.

Spread: The distribution is relatively wide, indicating that there is a significant variation in the difference between actual and estimated travel times. This suggests that OSRM's accuracy can vary depending on various factors.

19. Scatterplot for Segment Actual Time vs OSRM Distance

Code:

Output:



Insights:

Overall Relationship:

Positive Correlation: There appears to be a positive correlation between Segment Actual Time and Segment OSRM Distance. This suggests that as the actual travel time increases, the OSRM estimated distance also tends to increase.

Route Type Differences:

Clustering: The data points seem to cluster into two distinct groups, likely corresponding to the two route types: Carting and FTL.

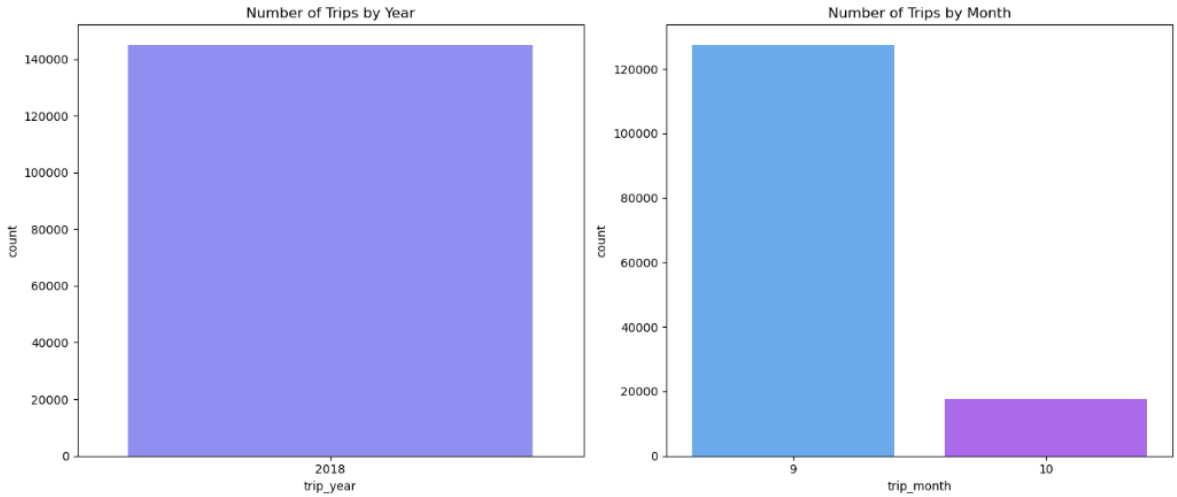
Spread: The Carting route type appears to have a tighter cluster of data points, indicating less variability in travel times and distances compared to the FTL route type.

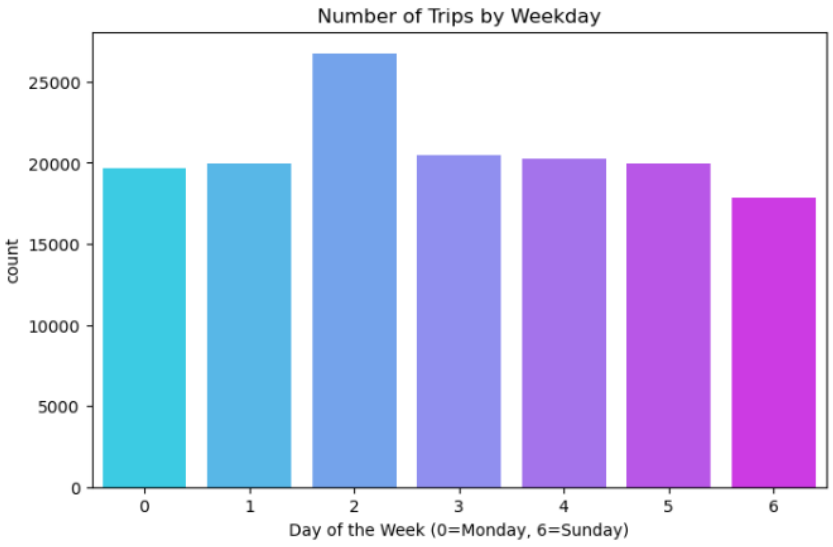
20. Barplot for trips by year and month

Code:

```
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
sns.countplot(x='trip_year', data=data, palette='cool')
plt.title("Number of Trips by Year")

plt.subplot(1, 2, 2)
sns.countplot(x='trip_month', data=data, palette='cool')
plt.title("Number of Trips by Month")

plt.tight_layout()
plt.show()

# Trips by weekday
plt.figure(figsize=(8, 5))
sns.countplot(x='trip_weekday', data=data, palette='cool')
plt.title("Number of Trips by Weekday")
plt.xlabel("Day of the Week (0=Monday, 6=Sunday)")
plt.show()
```

Output:





Insights:

- **Weekday Dominance:** Most trips occur on weekdays, suggesting a strong commuter base.
- **Seasonal Variation:** Trip numbers fluctuate monthly, with a peak in September.
- **Weekend Usage:** Despite being less popular, there's still considerable weekend usage.

Recommendations:

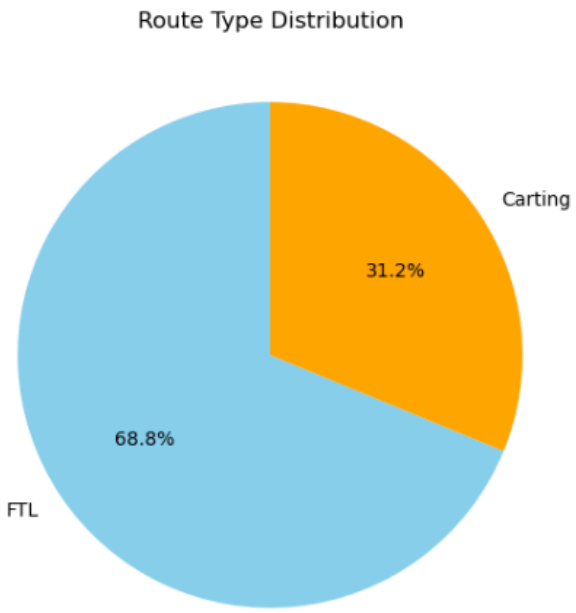
- **Optimize for Peak Times:** Focus on improving service during peak months and weekdays.
- **Encourage Weekend Usage:** Implement strategies like weekend packages or promotions.

21. Pie-Chart for Route Type Distribution

Code:

```
plt.figure(figsize=(6, 6))
data['route_type'].value_counts().plot.pie(autopct='%1.1f%%', colors=['skyblue', 'orange'], startangle=90)
plt.title("Route Type Distribution")
plt.ylabel("")
plt.show()
```

Output:



Insights and Recommendations:

- **Route Type Distribution:**
  - **FTL:** 68.8%
  - **Carting:** 31.2%

Implications:

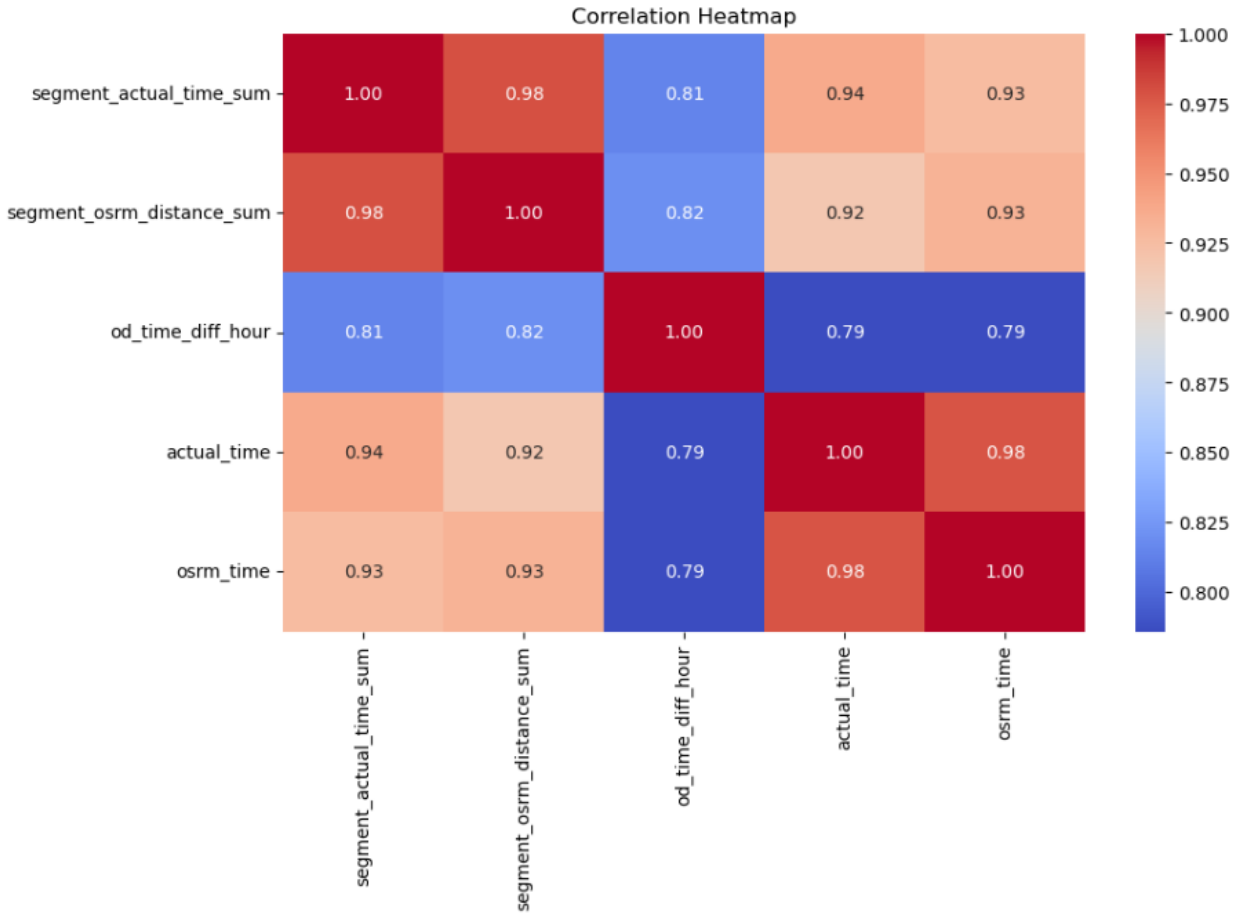
- **Diverse Operations:** The company handles a mix of large-scale (FTL) and smaller (Carting) shipments.
- **Strategic Planning:** Balancing resources and optimizing operations for both types is crucial.
- **Customer Focus:** Understanding and catering to the specific needs of customers using each route type is essential.

22. Correlation Heatmap

Code:

```
# Heatmap for correlation
plt.figure(figsize=(10, 6))
corr = data[['segment_actual_time_sum', 'segment_osrm_distance_sum', 'od_time_diff_hour', 'actual_time',
'osrm_time']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Correlation Heatmap")
plt.show()
```

Output:



Insights:

Strong Positive Correlations:

- **Time and Distance:** There is a very strong positive correlation between segment_actual_time_sum and segment_osrm_distance_sum. This suggests that longer distances generally lead to longer travel times.
- **Actual and Estimated Times:** actual_time and osrm_time are highly correlated, indicating that OSRM's estimates are generally accurate.

Moderate Positive Correlations:

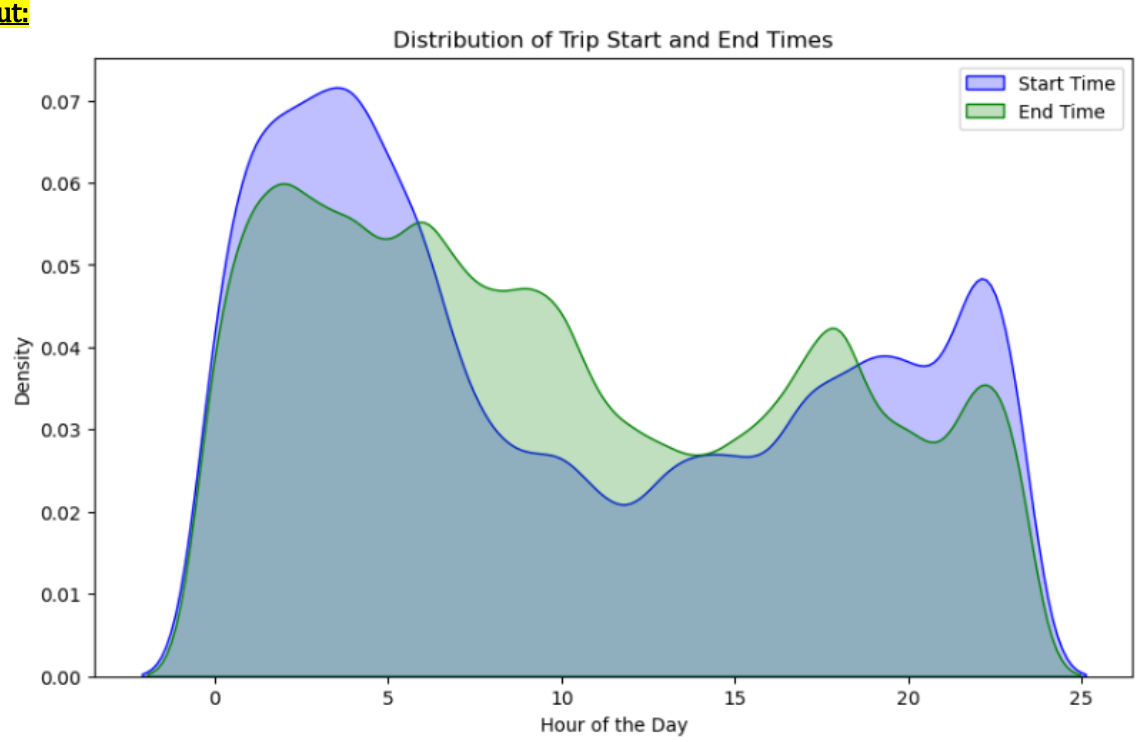
- **Time Difference and Other Variables:** od_time_diff_hour has moderate positive correlations with segment_actual_time_sum, segment_osrm_distance_sum, actual_time, and osrm_time. This suggests that larger differences between actual and estimated times are often associated with longer travel times and distances.

23. KDE plot for trip start and end times

Code:

```
plt.figure(figsize=(10, 6))
sns.kdeplot(data['od_start_time'].dt.hour, shade=True, color='blue', label='Start Time')
sns.kdeplot(data['od_end_time'].dt.hour, shade=True, color='green', label='End Time')
plt.title("Distribution of Trip Start and End Times")
plt.xlabel("Hour of the Day")
plt.ylabel("Density")
plt.legend()
plt.show()
```

Output:



Insights:

Peak Hours:

- **Morning Rush:** There's a clear peak in trip starts around 7-9 AM, indicating a significant number of trips beginning during the morning rush hour.
- **Evening Rush:** Another peak is observed around 5-7 PM, likely corresponding to the evening rush hour when people commute home or engage in evening activities.

Off-Peak Hours:

- **Early Morning and Late Night:** The lowest density of trip starts and ends occurs during the early morning hours (before 5 AM) and late night hours (after 11 PM). This suggests lower demand for rides during these times.

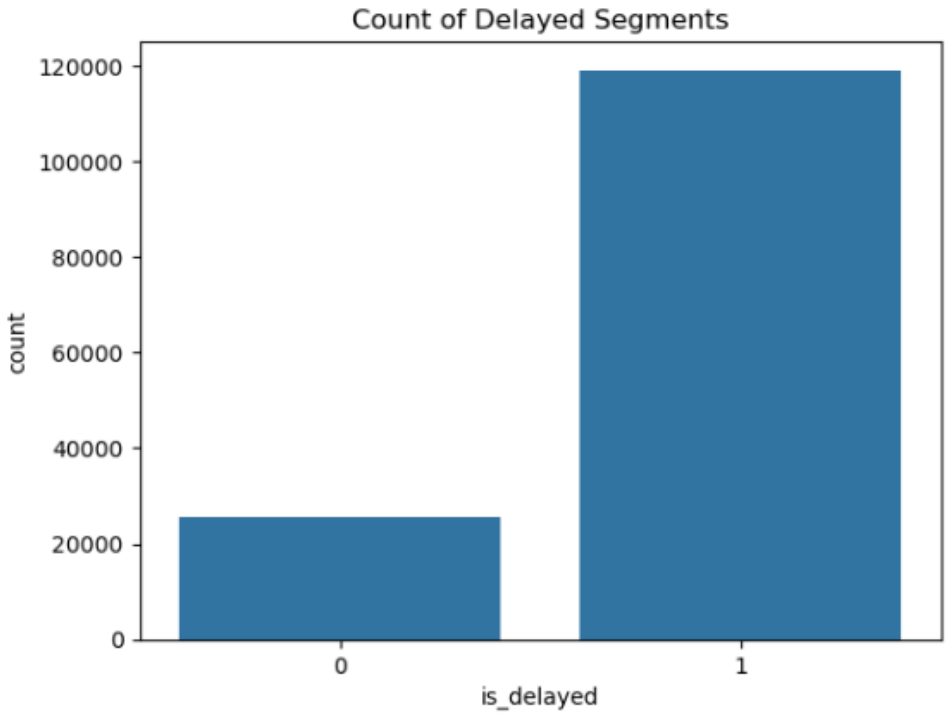
Overlap Between Start and End Times:

- The overlapping areas in the plot indicate times when both trip starts and ends are frequent. This could be due to various factors like short-distance trips, multiple trips per day, or people returning home after work.

24. Count of delayed Segment:

Code:

```
data['is_delayed'] = (data['actual_time'] > data['osrm_time'] * 1.5).astype(int)
plt.title('Count of Delayed Segments')
plt.show()
```



Insights:

- **Preponderance of Delays:** The bar chart clearly shows that the number of delayed segments is significantly higher than the number of non-delayed segments.
- **Potential Operational Issues:** The large number of delays suggests potential operational challenges or external factors affecting the service.

Possible Interpretations:

- **Traffic Congestion:** Heavy traffic conditions, especially during peak hours, can lead to delays.
- **Adverse Weather Conditions:** Inclement weather like rain, snow, or fog can significantly impact travel times and cause delays.
- **Vehicle Maintenance Issues:** Mechanical breakdowns or maintenance issues can lead to delays.

Recommendations:

Real-Time Traffic Monitoring: Implementing real-time traffic monitoring systems can help identify potential delays and reroute vehicles to avoid congested areas.
Predictive Analytics: Using historical data and machine learning techniques to predict potential delays can help proactively adjust schedules and inform customers.
Robust Maintenance Programs: Regular vehicle maintenance can help minimize breakdowns and reduce delays.

25. Measuring skewness for all numerical features

Code:

```
numerical_features = data.select_dtypes(include=['float64', 'int64']).columns
skewness = data[numerical_features].apply(lambda x: x.skew()).sort_values(ascending=False)

print("Skewness of numerical features:")
print(skewness)
```

Output:

```
Skewness of numerical features:
segment_factor                  47.372766
segment_osrm_distance           26.585363
segment_avg_time                24.812078
segment_osrm_time               19.639634
factor                          17.490811
segment_actual_time             16.827413
distance_efficiency             10.927730
distance_deviation               2.471980
time_efficiency                  2.400902
time_deviation                   2.375675
actual_time                      2.068065
segment_osrm_time_sum            2.065670
osrm_distance                    2.048236
osrm_time                        2.045166
cutoff_factor                    1.992094
actual_distance_to_destination   1.991105
segment_osrm_distance_sum        1.220784
segment_actual_time_sum          1.217751
start_scan_to_end_scan           1.110624
od_time_diff_hour                1.093335
monthly_trip_count              -2.325356
dtype: float64
```

Insights:

Skewness measures the asymmetry of the data distribution. A skewness value close to **0** indicates a symmetric distribution, while positive or negative values indicate skewed distributions. Here's an analysis based on the skewness values provided

Highly Skewed Features (Skewness > 2): Features like segment_factor, segment_avg_time, and distance_efficiency show extreme positive skewness, indicating significant outliers or rare extreme values. These may require transformations (e.g., log or Box-Cox) for better analysis.

Moderately Skewed Features (1 < Skewness ≤ 2): Features such as actual_time, osrm_distance, and od_time_diff_hour exhibit slight right-tailed distributions, suggesting most values are clustered around lower ranges with a few larger outliers.

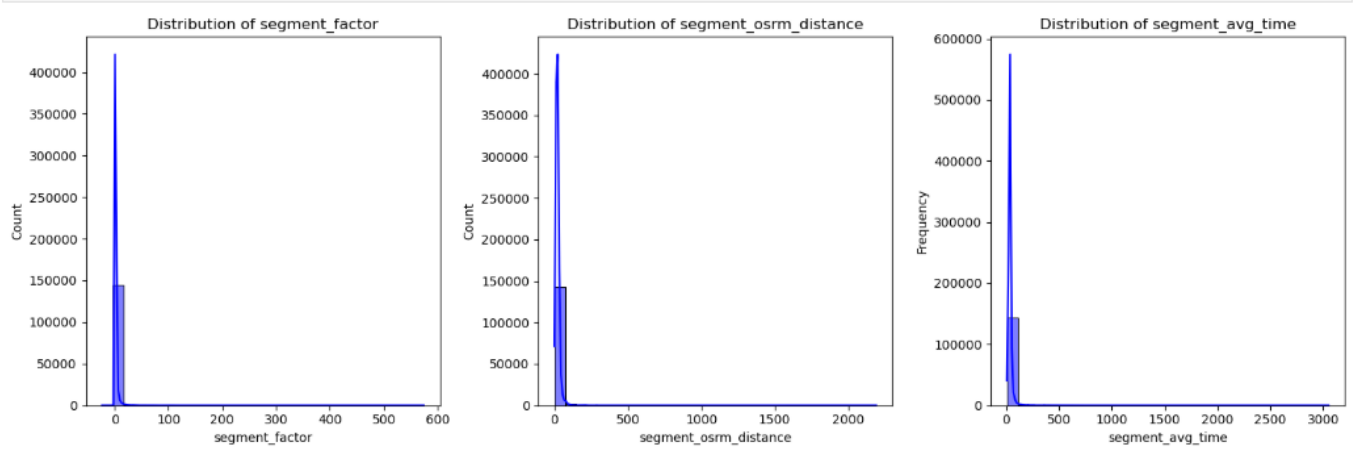
Negatively Skewed Feature: monthly_trip_count shows negative skewness, indicating a few months with significantly lower trip counts, likely reflecting seasonal or operational variations.

26. Visualize the top 3 most skewed features

Code:

```
top_skewed = skewness.head(3).index

plt.figure(figsize=(15, 5))
for i, feature in enumerate(top_skewed, 1):
    plt.subplot(1, 3, i)
    sns.histplot(data[feature], kde=True, bins=30, color='blue')
    plt.title(f"Distribution of {feature}")
plt.xlabel(feature)
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

Output:



Final Conclusion And Recommendation:

Conclusion:

Through detailed analysis and feature engineering, I evaluated the efficiency and accuracy of OSRM-predicted trip times and distances compared to actual values. Key insights include:

- **Time and Distance Efficiency:** OSRM often underestimates actual trip durations, as evident from the significant mean difference (approximately 205 seconds) and paired t-test results, indicating room for improvement in time predictions.
- **Deviation Analysis:** High deviations between OSRM-predicted and actual values suggest inconsistencies in trip planning, particularly for certain segments and route types.
- **Segment and Route Performance:** Analysis of segment-level metrics and route types revealed significant variability in trip efficiency, indicating that some routes or regions may experience delays due to traffic or other factors.
- **Seasonal Trends:** Negative skewness in monthly trip counts highlights potential seasonality or operational fluctuations, which can impact resource planning.
- **Customer Experience Metrics:** Time deviation and delay metrics identified specific trips or routes where customer dissatisfaction might arise due to delays exceeding expected thresholds.

Recommendations:

1. **Enhance Prediction Models:**
   Improve OSRM's prediction algorithms by incorporating real-time traffic data, historical delays, and contextual factors (e.g., weather, road closures) to reduce underestimation biases.
2. **Monitor and Address Inefficiencies:**
   Focus on routes and regions with the highest deviations in time and distance efficiency. Implement traffic-aware routing and proactive monitoring for these areas.
3. **Seasonal Resource Allocation:**
   Plan resources based on identified seasonal trends in trip volumes (monthly_trip_count) to minimize delays during peak months and optimize staffing or fleet availability.
4. **Customer-Centric Approach:**
   Flag trips exceeding 1.5x predicted times (is_delayed) and notify users proactively. Offering real-time updates or estimated delays can improve transparency and customer satisfaction.
5. **Periodic Audits:**
   Conduct periodic audits of OSRM predictions against actual trip metrics to continuously refine the system's accuracy and ensure operational reliability.

By implementing these recommendations, the organization can improve operational efficiency, enhance customer satisfaction, and optimize resource utilization effectively.

Jupyter Notebook Analysis

For a detailed view of the full analysis, including code, visualizations please refer to the complete Jupyter notebook available in the PDF format. The notebook documents each step of the analysis process, from data exploration to the final recommendations.

You can access the Jupyter notebook PDF through the **following link**:

Delhivery_Analysis