

DELHIVERY DATA ANALYSIS -- BY SUHASINI MOTTANNAVAR

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
! wget "https://d2beiqkhq929f0.cloudfront.net/public_assets/assets/000/001/551/original/delhivery_data.csv?1642751181"
```

```

--2025-06-10 17:06:18--  https://d2beiqkhq929f0.cloudfront.net/public_assets/assets/000/001/551/original/delhivery_data.csv?1642751181
Resolving d2beiqkhq929f0.cloudfront.net (d2beiqkhq929f0.cloudfront.net)... 13.224.9.24, 13.224.9.181, 13.224.9.129, ...
Connecting to d2beiqkhq929f0.cloudfront.net (d2beiqkhq929f0.cloudfront.net)|13.224.9.24|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 55617130 (53M) [text/plain]
Saving to: 'delhivery_data.csv?1642751181.4'

delhivery_data.csv? 100%[=====] 53.04M 188MB/s in 0.3s

2025-06-10 17:06:18 (188 MB/s) - 'delhivery_data.csv?1642751181.4' saved [55617130/55617130]
```

```
df = pd.read_csv("delhivery_data.csv?1642751181")
```

Data cleaning and exploration:

```
df.head()
```

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

5 rows x 24 columns

```
df.shape
```

```
(144867, 24)
```

```
df.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
```

df.describe()

	start_scan_to_end_scan	cutoff_factor	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	factor	se
count	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	
mean	961.262986	232.926567	234.073372	416.927527	213.868272	284.771297	2.120107	
std	1037.012769	344.755577	344.990009	598.103621	308.011085	421.119294	1.715421	
min	20.000000	9.000000	9.000045	9.000000	6.000000	9.008200	0.144000	
25%	161.000000	22.000000	23.355874	51.000000	27.000000	29.914700	1.604264	
50%	449.000000	66.000000	66.126571	132.000000	64.000000	78.525800	1.857143	
75%	1634.000000	286.000000	286.708875	513.000000	257.000000	343.193250	2.213483	
max	7898.000000	1927.000000	1927.447705	4532.000000	1686.000000	2326.199100	77.387097	

```
# Check for missing values in each column
missing_values = df.isnull().sum()
```

```
# Display the number of missing values in each column
print("Number of missing values in each column:")
print(missing_values)
```

```
Number of missing values in each column:
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
```

```
# 1. Handle missing values
# We'll fill missing source_name and destination_name with "Unknown"
df['source_name'].fillna("Unknown", inplace=True)
df['destination_name'].fillna("Unknown", inplace=True)
structure_summary
```

```
<ipython-input-151-4e586f579671>:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained ass
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting v

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col]

df['source_name'].fillna("Unknown", inplace=True)
<ipython-input-151-4e586f579671>:4: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained ass
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting v

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col]
```

	count	unique	top	freq	mean	min	25%	50%
data	144867	2	training	104858	NaN	NaN	NaN	NaN
trip_creation_time	144867	NaN	NaN	NaN	2018-09-22 13:34:23.659819264	2018-09-12 00:00:16.535741	2018-09-17 03:20:51.775845888	2018-09-22 04:24:27.93276492
route_schedule_uuid	144867	1504	thanos::sroute:4029a8a2-6c74-4b7e-a6d8-f9e069f...	1812	NaN	NaN	NaN	NaN
route_type	144867	2	FTL	99660	NaN	NaN	NaN	NaN
trip_uuid	144867	14817	trip-153759210483476123	101	NaN	NaN	NaN	NaN
source_center	144867	1508	IND000000ACB	23347	NaN	NaN	NaN	NaN
source_name	144867	1499	Gurgaon_Bilaspur_HB (Haryana)	23347	NaN	NaN	NaN	NaN
destination_center	144867	1481	IND000000ACB	15192	NaN	NaN	NaN	NaN
destination_name	144867	1469	Gurgaon_Bilaspur_HB (Haryana)	15192	NaN	NaN	NaN	NaN
od_start_time	144867	NaN	NaN	NaN	2018-09-22 18:02:45.855230720	2018-09-12 00:00:16.535741	2018-09-17 08:05:40.886155008	2018-09-22 08:53:00.11665612
od_end_time	144867	NaN	NaN	NaN	2018-09-23 10:04:31.395393024	2018-09-12 00:50:10.814399	2018-09-18 01:48:06.410121984	2018-09-23 03:13:03.52021296
start_scan_to_end_scan	144867.0	NaN	NaN	NaN	961.262986	20.0	161.0	449
is_cutoff	144867	2	True	118749	NaN	NaN	NaN	NaN
cutoff_factor	144867.0	NaN	NaN	NaN	232.926567	9.0	22.0	66
cutoff_timestamp	141438	NaN	NaN	NaN	2018-09-23 03:43:41.794807552	2018-09-12 00:10:27	2018-09-17 19:52:04.750000128	2018-09-23 22:02:5
actual_distance_to_destination	144867.0	NaN	NaN	NaN	234.073372	9.000045	23.355874	66.12657
actual_time	144867.0	NaN	NaN	NaN	416.927527	9.0	51.0	132
osrm_time	144867.0	NaN	NaN	NaN	213.868272	6.0	27.0	64
osrm_distance	144867.0	NaN	NaN	NaN	284.771297	9.0082	29.9147	78.525
factor	144867.0	NaN	NaN	NaN	2.120107	0.144	1.604264	1.85714
segment_actual_time	144867.0	NaN	NaN	NaN	36.196111	-244.0	20.0	29
segment_osrm_time	144867.0	NaN	NaN	NaN	18.507548	0.0	11.0	17
segment_osrm_distance	144867.0	NaN	NaN	NaN	22.82902	0.0	12.0701	23.51
segment_factor	144867.0	NaN	NaN	NaN	2.218368	-23.444444	1.347826	1.68421

Next steps: [Generate code with structure\\_summary](#) [View recommended plots](#) [New interactive sheet](#)

```
# 2. Convert time columns to pandas datetime
time_columns = ['trip_creation_time', 'od_start_time', 'od_end_time', 'cutoff_timestamp']
for col in time_columns:
    df[col] = pd.to_datetime(df[col], errors='coerce')

data_types
```



0

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

```
# 3. Analyze structure & characteristics
structure_summary = df.describe(include='all').transpose()
data_types = df.dtypes
unique_counts = df.nunique()

unique_counts
```



0

data	2
trip_creation_time	14817
route_schedule_uuid	1504
route_type	2
trip_uuid	14817
source_center	1508
source_name	1499
destination_center	1481
destination_name	1469
od_start_time	26369
od_end_time	26369
start_scan_to_end_scan	1915
is_cutoff	2
cutoff_factor	501
cutoff_timestamp	90130
actual_distance_to_destination	144515
actual_time	3182
osrm_time	1531
osrm_distance	138046
factor	45641
segment_actual_time	747
segment_osrm_time	214
segment_osrm_distance	113799
segment_factor	5675

dtype: int64

```
# Create segment_key as combination of trip_uuid, source_center, destination_center
df['segment_key'] = df['trip_uuid'] + "_" + df['source_center'] + "_" + df['destination_center']
```

```
# Group by segment_key and compute cumulative sums for segment-specific columns
df['segment_actual_time_sum'] = df.groupby('segment_key')['segment_actual_time'].cumsum()
df['segment_osrm_distance_sum'] = df.groupby('segment_key')['segment_osrm_distance'].cumsum()
df['segment_osrm_time_sum'] = df.groupby('segment_key')['segment_osrm_time'].cumsum()
```


```
# Define aggregation rules for each column in a dictionary
```

```
create_segment_dict = {
    'trip_creation_time': 'first',
    'route_schedule_uuid': 'first',
    'route_type': 'first',
    'trip_uuid': 'first',
    'source_center': 'first',
    'source_name': 'first',
    'destination_center': 'first',
    'destination_name': 'first',
    'od_start_time': 'first',
    'od_end_time': 'last',
    'start_scan_to_end_scan': 'sum',
    'is_cutoff': 'last',
    'cutoff_factor': 'sum',
    'cutoff_timestamp': 'last',
    'actual_distance_to_destination': 'sum',
    'actual_time': 'sum',
    'osrm_time': 'sum',
    'osrm_distance': 'sum',
    'factor': 'mean',
    'segment_actual_time': 'sum',
    'segment_osrm_time': 'sum',
    'segment_osrm_distance': 'sum',
    'segment_factor': 'mean',
    'segment_actual_time_sum': 'last',
    'segment_osrm_distance_sum': 'last',
    'segment_osrm_time_sum': 'last'
}
```

```
# Group by segment_key and apply aggregation
segment_df = df.groupby('segment_key').agg(create_segment_dict).reset_index()
```

```
# Sort by segment_key and then by od_end_time
segment_df.sort_values(by=['segment_key', 'od_end_time'], ascending=[True, True], inplace=True)

segment_df.head()
```



	segment_key	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center
0	trip-153671041653548748_IND209304AAA_IND000000ACB	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND209304AAA
1	trip-153671041653548748_IND462022AAA_IND209304AAA	2018-09-12 00:00:16.535741	thanos::sroute:d7c989ba-a29b-4a0b-b2f4-288cdc6...	FTL	trip-153671041653548748	IND462022AAA
2	trip-153671042288605164_IND561203AAB_IND562101AAA	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND561203AAB
3	trip-153671042288605164_IND572101AAA_IND561203AAB	2018-09-12 00:00:22.886430	thanos::sroute:3a1b0ab2-bb0b-4c53-8c59-eb2a2c0...	Carting	trip-153671042288605164	IND572101AAA
4	trip-153671043369099517_IND000000ACB_IND160002AAC	2018-09-12 00:00:33.691250	thanos::sroute:de5e208e-7641-45e6-8100-4d9fb1e...	FTL	trip-153671043369099517	IND000000ACB

5 rows × 27 columns

Feature Engineering:

```
# 1. Time difference between od_start_time and od_end_time
df['od_time_diff_hour'] = (df['od_end_time'] - df['od_start_time']).dt.total_seconds() / 3600
df.drop(columns=['od_start_time', 'od_end_time'], inplace=True)

# 2. Extract features from Destination Name
# Expected format: City-place-code (State)
def parse_destination(dest):
    try:
        # Split on ' (' to separate state
        city_place, state = dest.split('(')[0], dest.split('(')[1].strip(' ')
        # Split city_place on '_' to get components
        parts = city_place.split('_')
        city = parts[0]
        place = parts[1] if len(parts) > 2 else ''
        code = parts[-1]
        state = state.strip(' ')
        return pd.Series({'dest_city': city, 'dest_place': place, 'dest_code': code, 'dest_state': state})
    except:
        return pd.Series({'dest_city': '', 'dest_place': '', 'dest_code': '', 'dest_state': ''})

# Apply parsing function
dest_features = df['destination_name'].apply(parse_destination)
df = pd.concat([df, dest_features], axis=1)

# 3. Extract features from Source Name
# Same format as destination
def parse_source(source):
    try:
        city_place, state = source.split(' (')
        parts = city_place.split('_')
        city = parts[0]
        place = parts[1] if len(parts) > 2 else ''
        code = parts[-1]
        state = state.strip(' ')
        return pd.Series({'source_city': city, 'source_place': place, 'source_code': code, 'source_state': state})
    except:
        # Indent the except block and its contents to align with the try block
        return pd.Series({'source_city': '', 'source_place': '', 'source_code': '', 'source_state': ''})

# Apply parsing function
source_features = df['source_name'].apply(parse_source)
df = pd.concat([df, source_features], axis=1)

# 4. Extract features from trip_creation_time
df['trip_month'] = df['trip_creation_time'].dt.month
df['trip_year'] = df['trip_creation_time'].dt.year
df['trip_day'] = df['trip_creation_time'].dt.day
```

```
df['trip_weekday'] = df['trip_creation_time'].dt.weekday
df['trip_hour'] = df['trip_creation_time'].dt.hour
```

```
# Preview the new features
print(df[['od_time_diff_hour', 'dest_city', 'dest_place', 'dest_state',
          'source_city', 'source_place', 'source_state', 'trip_month', 'trip_year',
          'trip_day', 'trip_weekday', 'trip_hour']].head())
```

```
↗
   od_time_diff_hour dest_city dest_place dest_state source_city source_place \
0          1.436894  Khambhat  MotvdDPP   Gujarat      Anand      VUNagar
1          1.436894  Khambhat  MotvdDPP   Gujarat      Anand      VUNagar
2          1.436894  Khambhat  MotvdDPP   Gujarat      Anand      VUNagar
3          1.436894  Khambhat  MotvdDPP   Gujarat      Anand      VUNagar
4          1.436894  Khambhat  MotvdDPP   Gujarat      Anand      VUNagar

   source_state trip_month trip_year trip_day trip_weekday trip_hour
0      Gujarat         9      2018      20         3         2
1      Gujarat         9      2018      20         3         2
2      Gujarat         9      2018      20         3         2
3      Gujarat         9      2018      20         3         2
4      Gujarat         9      2018      20         3         2
```

## ↘ In-depth analysis

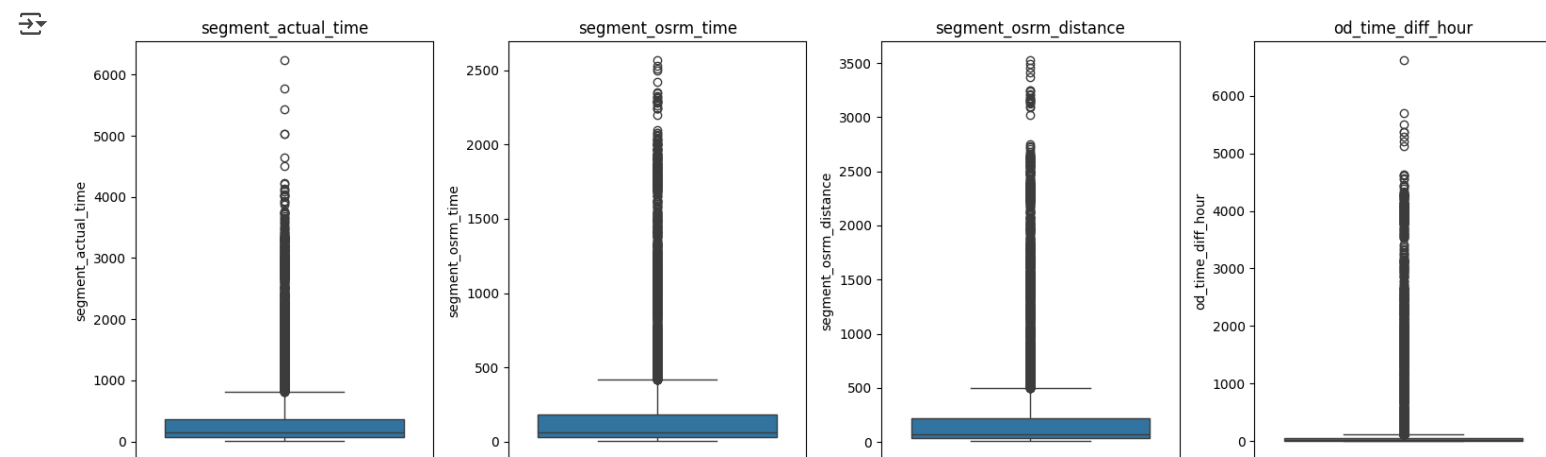
```
# Define aggregation strategy
create_trip_dict = {
    'route_schedule_uuid': 'first',
    'route_type': 'first',
    'source_center': 'first',
    'destination_center': 'last',
    'source_name': 'first',
    'destination_name': 'last',
    'trip_creation_time': 'first',
    'segment_actual_time': 'sum',
    'segment_osrm_time': 'sum',
    'segment_osrm_distance': 'sum',
    'segment_factor': 'mean',
    'od_time_diff_hour': 'sum', # if created
    'cutoff_timestamp': 'last',
    'cutoff_factor': 'sum'
}
```

```
# Group by trip_uuid
trip_df = df.groupby('trip_uuid').agg(create_trip_dict).reset_index()
```

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

```
num_cols = ['segment_actual_time', 'segment_osrm_time', 'segment_osrm_distance', 'od_time_diff_hour']
```

```
plt.figure(figsize=(16, 5))
for i, col in enumerate(num_cols):
    plt.subplot(1, len(num_cols), i + 1)
    sns.boxplot(y=trip_df[col])
    plt.title(col)
plt.tight_layout()
plt.show()
```



### 1. segment\_actual\_time

The actual time taken to travel a segment.

#### Insights:

- The median (middle line of the box) is relatively low compared to the extreme upper outliers.
- There's a heavy right skew (long whisker on the top), meaning many segments took a lot more time than the majority.
- Outliers are present far above the interquartile range (IQR), indicating segments with unusually high travel time.

### 2. segment\_osrm\_time

Estimated time to traverse a segment according to the OSRM (Open Source Routing Machine).

#### Insights:

- Distribution is more compact compared to actual time.
- Still, there are noticeable outliers, but fewer and less extreme than in the actual time.
- This suggests OSRM estimates are more consistent but possibly underestimate real-world traffic effects or delays.

### 3. segment\_osrm\_distance

Distance of the segment as computed by OSRM.

#### Insights:

- As expected, distance is more stable and less variable.
- Some outliers exist, indicating segments that are significantly longer than most.
- This variable has less noise compared to time-related variables, which makes sense since distance is a fixed attribute unlike time.

### 4. od\_time\_diff\_hour

Time difference (in hours) between origin and destination—perhaps total journey duration. **Insights:**

- This variable is highly skewed as well, with a few extremely large values.
- Could indicate data quality issues or specific edge cases like very long delays.
- The bulk of the data lies in a much smaller range.

```
def iqr_filter(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] >= lower_bound) & (data[column] <= upper_bound)]

for col in num_cols:
    trip_df = iqr_filter(trip_df, col)

cat_cols = ['route_type']

# Fill NA to avoid errors
trip_df[cat_cols] = trip_df[cat_cols].fillna('Unknown')

# Apply one-hot encoding
trip_df = pd.get_dummies(trip_df, columns=cat_cols, drop_first=True)

from sklearn.preprocessing import MinMaxScaler

# Select numeric columns for scaling
scale_cols = ['segment_actual_time', 'segment_osrm_time', 'segment_osrm_distance', 'od_time_diff_hour', 'cutoff_factor']

scaler = MinMaxScaler()
trip_df[scale_cols] = scaler.fit_transform(trip_df[scale_cols])
```

## ~ Hypothesis Testing:

```
# Aggregate segment-level values by trip_uuid
segment_agg = segment_df.groupby('trip_uuid').agg({
    'segment_actual_time': 'sum',
    'segment_osrm_time': 'sum',
    'segment_osrm_distance': 'sum'
}).reset_index()
```

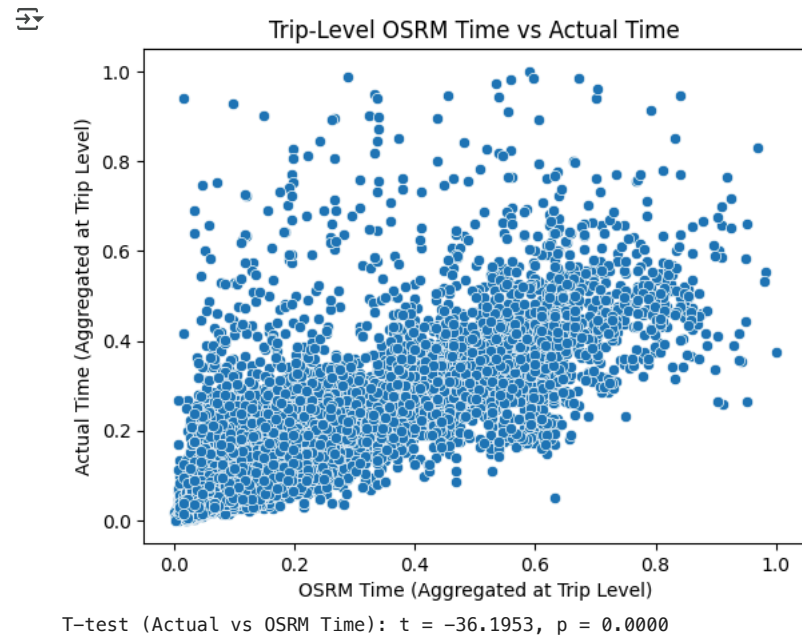


```
merged = pd.merge(trip_df, segment_agg, on='trip_uuid', suffixes=('_trip', '_segment'))
```

```
from scipy.stats import ttest_rel
```

```
# Visual
sns.scatterplot(x=merged['segment_osrm_time_trip'], y=merged['segment_actual_time_trip'])
plt.xlabel("OSRM Time (Aggregated at Trip Level)")
plt.ylabel("Actual Time (Aggregated at Trip Level)")
plt.title("Trip-Level OSRM Time vs Actual Time")
plt.show()

# Statistical test
t_stat, p_val = ttest_rel(merged['segment_actual_time_trip'], merged['segment_osrm_time_trip'])
print(f"T-test (Actual vs OSRM Time): t = {t_stat:.4f}, p = {p_val:.4f}")
```



This scatter plot visualizes the relationship between OSRM-predicted travel times and actual travel times, both aggregated at the trip level and normalized (scaled between 0 and 1).

#### 1. Positive Correlation:

- There's a general upward trend, meaning: higher OSRM time estimates tend to be associated with higher actual travel times.
- But the spread of points indicates substantial variability.

#### 2. OSRM Underestimation:

- Many data points lie above the diagonal  $y = x$  line (implied).
- This indicates that in many cases, actual times are greater than predicted times — OSRM underestimates real-world travel durations.

#### 3. Noise and Variance:

- The scatter is dense and spread out, especially for mid-range OSRM values.
- This implies high variance and lower predictive precision in some ranges.

#### 4. Low values clustered:

- A lot of trips are clustered in the 0.1 to 0.5 range on both axes, which might indicate that many trips are short or quick.

**T-statistic: -36.1953** → A large negative value indicates that the mean of actual times is significantly greater than OSRM times.

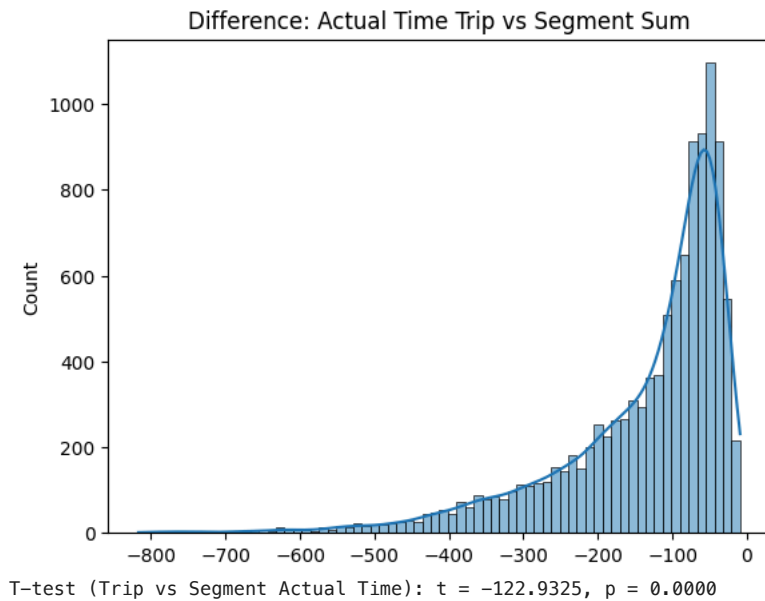
**p-value = 0.0000 (very small)** → The difference is statistically significant — we reject the null hypothesis that the means are equal.

#### Conclusion from T-Test:

- There is a statistically significant difference between actual and predicted trip times.
- OSRM consistently underestimates travel durations, and this underestimation is not due to random chance.

```
# Visual
sns.histplot(merged['segment_actual_time_trip'] - merged['segment_actual_time_segment'], kde=True)
plt.title("Difference: Actual Time Trip vs Segment Sum")
plt.show()

# Paired t-test
t_stat, p_val = ttest_rel(merged['segment_actual_time_trip'], merged['segment_actual_time_segment'])
print(f"T-test (Trip vs Segment Actual Time): t = {t_stat:.4f}, p = {p_val:.4f}")
```



This histogram visualizes the difference between actual trip time and the sum of segment-level actual times.

#### 1. Most values are negative:

- The distribution is heavily skewed to the left.
- This means that, for most trips, the actual time recorded at the trip level is less than the sum of actual times of individual segments.

#### 2. Peak near -50 to 0:

- The mode (highest count) is in the range of -100 to 0, indicating that the typical difference is around -50 to -100 seconds.
- Suggests a systematic overestimation when summing up segment times.

#### 3. Long left tail:

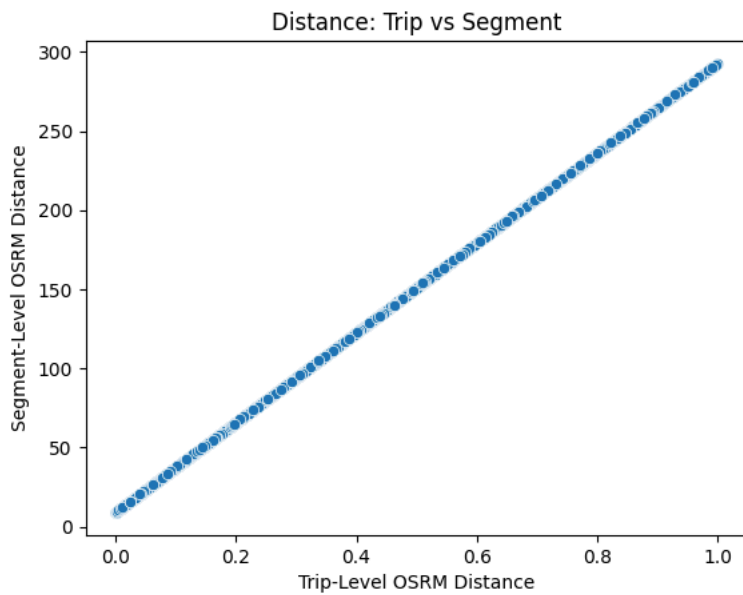
- Some differences are very large (up to -800), suggesting data quality issues, segment overlap, delays counted multiple times, or mismatches in time recording.

**T-statistic: -122.93** → A very large negative t-statistic means the mean of the trip-level actual time is significantly less than the sum of segment actual times.

**p-value = 0.0000** → The result is statistically significant, so we reject the null hypothesis that the means are the same.

```
# Visual
sns.scatterplot(x=merged['segment_osrm_distance_trip'], y=merged['segment_osrm_distance_segment'])
plt.xlabel("Trip-Level OSRM Distance")
plt.ylabel("Segment-Level OSRM Distance")
plt.title("Distance: Trip vs Segment")
plt.show()

# Paired t-test
t_stat, p_val = ttest_rel(merged['segment_osrm_distance_trip'], merged['segment_osrm_distance_segment'])
print(f"T-test (OSRM Distance Trip vs Segment): t = {t_stat:.4f}, p = {p_val:.4f}")
```



T-test (OSRM Distance Trip vs Segment):  $t = -118.8180$ ,  $p = 0.0000$

This scatter plot visualizes the relationship between trip-level OSRM distance and the sum of segment-level OSRM distances.

**1. Strong linear relationship:**

- The data points lie nearly perfectly along the diagonal line, which suggests a strong positive linear correlation between trip-level and segment-level OSRM distances.

**2. Near equality visually:**

- For most data points, the trip-level OSRM distance is almost equal to the sum of the segment-level distances, indicating consistency between the two.

**T-statistic: -118.8180** → A very large negative t-statistic, implying a statistically significant mean difference between trip and segment distances.

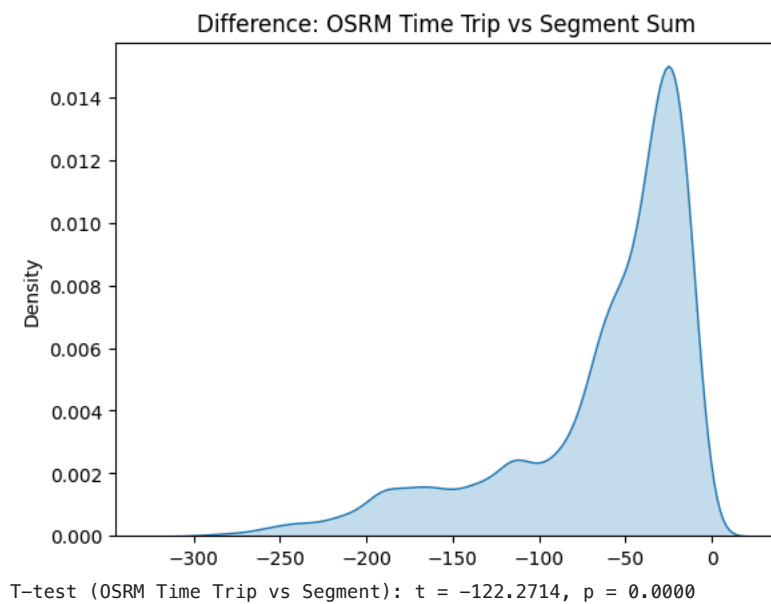
**p-value = 0.0000** → The p-value is virtually zero, meaning we reject the null hypothesis of equal means.

Even though the scatter plot looks perfectly aligned, the t-test is highly significant. This suggests that:

- There is a consistent, slight difference between trip-level and segment-level OSRM distances.
- Since the sample size is large, even a tiny, systematic difference can be detected as statistically significant.
- Distance aggregation from segments might include rounding, truncation, or path smoothing effects.
- Trip-level OSRM distance might use a simplified route, while segment-level includes full details.

```
# Visual
sns.kdeplot(merged['segment_osrm_time_trip'] - merged['segment_osrm_time_segment'], fill=True)
plt.title("Difference: OSRM Time Trip vs Segment Sum")
plt.show()

# Paired t-test
t_stat, p_val = ttest_rel(merged['segment_osrm_time_trip'], merged['segment_osrm_time_segment'])
print(f"T-test (OSRM Time Trip vs Segment): t = {t_stat:.4f}, p = {p_val:.4f}")
```



This graph presents a distribution of the difference between OSRM Trip-Level Time and the Sum of Segment-Level OSRM Times.

#### 1. Distribution shape:

- The KDE curve is unimodal and skewed to the left, with the peak near zero but noticeably shifted towards negative values.
- This suggests that in most trips, the sum of segment-level OSRM times is greater than the trip-level OSRM time.

#### 2. Range:

- Differences go as far back as -300+ units, indicating some substantial discrepancies.
- There are very few values close to or greater than zero, indicating almost always the trip time is less than the sum of segments.

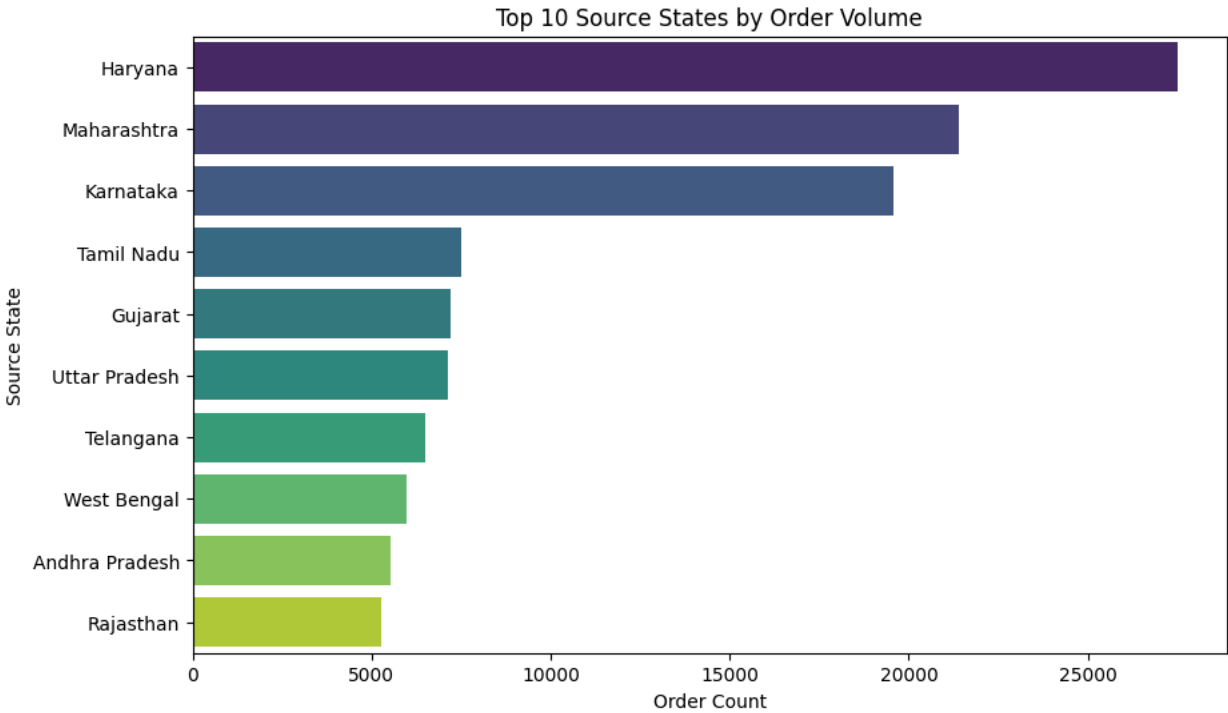
**T-statistic: -122.2714** → A very large negative t-statistic, which confirms that the mean difference is significantly less than zero.

**p-value = 0.0000:** → p-value effectively zero → extremely significant. We reject the null hypothesis that the mean difference is zero.

## ✓ Business Insights & Recommendations

```
# Top 10 Source States
top_source_states = df['source_state'].value_counts().nlargest(10)
fig1, ax1 = plt.subplots(figsize=(10, 6))
sns.barplot(x=top_source_states.values, y=top_source_states.index, palette='viridis', ax=ax1)
ax1.set_title("Top 10 Source States by Order Volume")
ax1.set_xlabel("Order Count")
ax1.set_ylabel("Source State")
```

```
<ipython-input-182-0d5379050755>:4: FutureWarning:
    Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `leg
sns.barplot(x=top_source_states.values, y=top_source_states.index, palette='viridis', ax=ax1)
Text(0, 0.5, 'Source State')
```



This bar chart visualizes the Top 10 Source States by Order Volume, showing the total number of orders originating from each state.

- Dominant state:
  - Haryana leads significantly. Possibly a major warehousing/logistics hub or a high-demand origin point.
- Clustered Middle States:
  - Maharashtra and Karnataka form a high-volume cluster.
  - Tamil Nadu, Gujarat, UP, and Telangana make up the middle tier.
- Long Tail:
  - West Bengal, Andhra Pradesh, and Rajasthan have smaller but non-negligible order volumes.
  - Suggests broader national distribution activity.

Conclusion

- Haryana is a strategic logistics state — investigate further into infrastructure or vendor concentration.
- Top 3 states account for a large portion of order volume — opportunity to optimize delivery routes here.
- Rajasthan and Andhra Pradesh are underperforming in comparison — could indicate opportunity or underutilization.

Double-click (or enter) to edit

```
df['source_city'].value_counts().head(11)
```



	count
source_city	
Gurgaon	23665
Bangalore	10104
Bhiwandi	9088
Pune	4269
Bengaluru	4237
Hyderabad	4023
Delhi	3587
Chandigarh	2957
Kolkata	2844
Surat	2362
Ahmedabad	1850

**dtype:** int64

Order Concentration by Source: Here are the top 10 cities (considering Bangalore and Bengaluru as the same city which provide "14341" counts) which provides majority of orders originate. These regions act as major dispatch hubs or central warehouses.

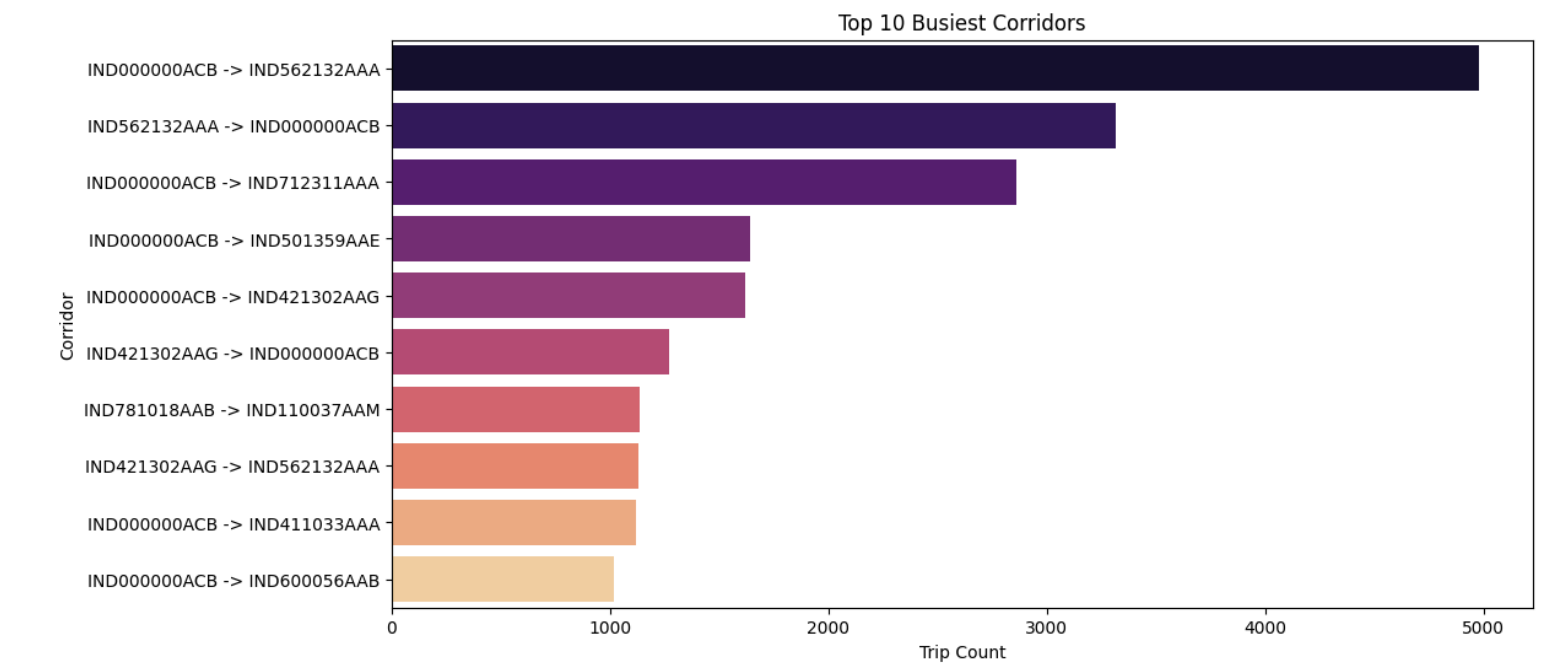
```
# Top 10 Corridors by Frequency
top_corridors = df['corridor'].value_counts().nlargest(10)
fig2, ax2 = plt.subplots(figsize=(12, 6))
sns.barplot(x=top_corridors.values, y=top_corridors.index, palette='magma', ax=ax2)
ax2.set_title("Top 10 Busiest Corridors")
ax2.set_xlabel("Trip Count")
ax2.set_ylabel("Corridor")

df['corridor'] = df['source_center'] + " -> " + df['destination_center']
top_corridors = df['corridor'].value_counts().head(10)
top_corridors
```

```
<ipython-input-185-626158f12729>:4: FutureWarning:
    Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `leg
sns.barplot(x=top_corridors.values, y=top_corridors.index, palette='magma', ax=ax2)

count
corridor
IND000000ACB -> IND562132AAA 4976
IND562132AAA -> IND000000ACB 3316
IND000000ACB -> IND712311AAA 2862
IND000000ACB -> IND501359AAE 1639
IND000000ACB -> IND421302AAG 1617
IND421302AAG -> IND000000ACB 1269
IND781018AAB -> IND110037AAM 1137
IND421302AAG -> IND562132AAA 1131
IND000000ACB -> IND411033AAA 1120
IND000000ACB -> IND600056AAB 1015

dtype: int64
```



This chart displays the Top 10 Busiest Corridors based on trip count, which likely represents the number of times goods or vehicles have traveled between specific origin-destination (OD) pairs.


- 1. **Dominance of IND000000ACB:**
  - This hub appears in 7 out of 10 corridors.
  - Acts as either source or destination frequently.
  - Likely a central distribution center or hub in the logistics network.
- 2. **Symmetrical Flow:**
  - The top 2 routes are mutual: ACB → AAA and AAA → ACB.
  - Suggests back-and-forth freight movement, possibly related to reusable assets like crates, vehicles, or inventory redistribution.
- 3. **Trip Distribution:**
  - Top 3 corridors have significantly higher trip counts (>3000).
  - There's a steep drop after the top 2–3, with the lowest corridor just above 1000 trips.

Conclusion




- Optimize the Top Corridors:
  - The high traffic on IND000000ACB routes may signal congestion or opportunity for route optimization, driver scheduling, or asset allocation.
- Balance Return Trips:

- The two-way flow in the top corridor suggests efficiency in return logistics — a good practice to extend to other corridors.
- Scalability Watch:
  - Monitor medium-tier corridors (Trip Counts ~1000–1500) for growth trends — they might become bottlenecks in the near future.

```
corridor_stats = df.groupby('corridor').agg({
    'segment_osrm_distance': 'mean',
    'segment_actual_time': 'mean',
    'segment_osrm_time': 'mean'
}).sort_values(by='segment_osrm_distance', ascending=False)
corridor_stats
```



	segment_osrm_distance	segment_actual_time	segment_osrm_time
corridor			
IND284403AAA -> IND474003AAA	223.265500	521.000000	208.000000
IND425412AAA -> IND424006AAA	109.161500	1093.000000	79.000000
IND173212AAA -> IND160002AAC	101.729600	191.000000	95.000000
IND743270AAA -> IND712311AAA	98.744900	1133.600000	73.800000
IND425409AAA -> IND424006AAA	94.560200	894.000000	74.000000
...	...	...	...
IND400016AAB -> IND400072AAB	5.592000	30.000000	6.000000
IND201005AAA -> IND201007AAA	5.494200	23.000000	7.000000
IND211002AAB -> IND211011AAA	5.245842	21.538462	3.307692
IND121004AAB -> IND121002AAA	5.177350	16.000000	5.500000
IND600032AAB -> IND600008AAC	5.023800	24.000000	4.000000



2783 rows x 3 columns

Next steps:

[Generate code with corridor\\_stats](#)

[View recommended plots](#)

[New interactive sheet](#)

Corridors with high actual time but low OSRM time may signal traffic, inefficiencies, or route planning issues.

Corridors with short distances and high time may indicate urban congestion.

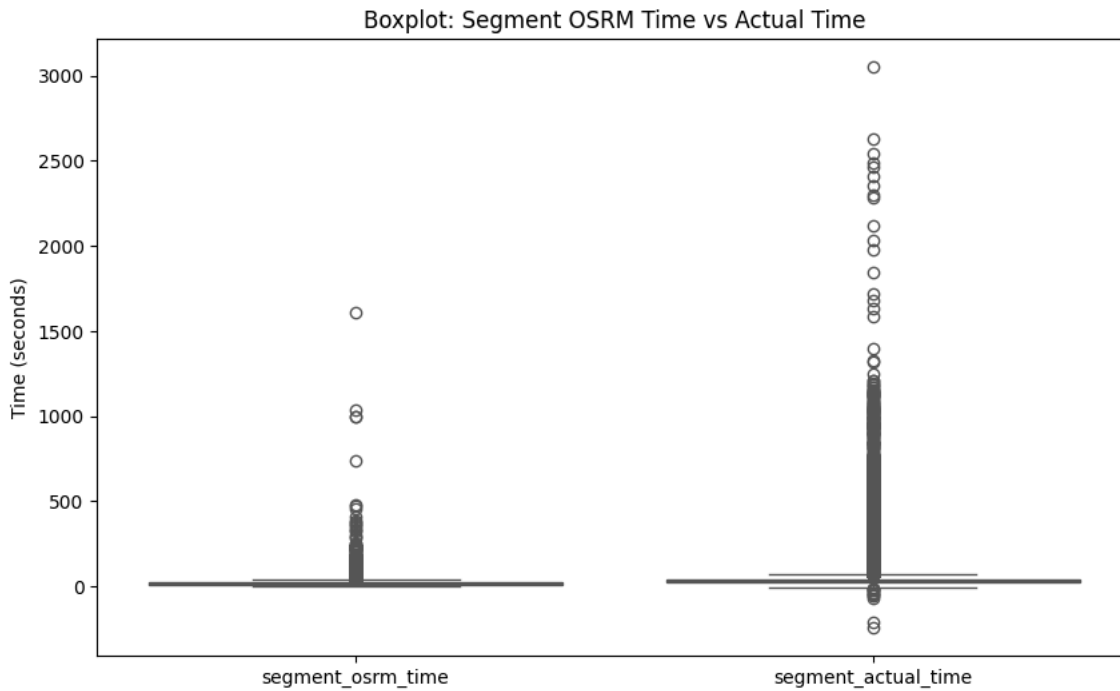
```
# Boxplot - OSRM vs Actual Time
fig3, ax3 = plt.subplots(figsize=(10, 6))
melted = df[['segment_osrm_time', 'segment_actual_time']].melt()
sns.boxplot(x='variable', y='value', data=melted, palette='Set2', ax=ax3)
ax3.set_title("Boxplot: Segment OSRM Time vs Actual Time")
ax3.set_ylabel("Time (seconds)")
ax3.set_xlabel("")
```



```
<ipython-input-186-6b497881f94a>:4: FutureWarning:
```

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `leg`

```
sns.boxplot(x='variable', y='value', data=melted, palette='Set2', ax=ax3)
Text(0.5, 0, '')
```



This is a boxplot comparing OSRM-predicted travel time (`segment_osrm_time`) with the actual travel time (`segment_actual_time`) for route segments.

The box represents the interquartile range (IQR) — the middle 50% of the data. The line inside the box is the median. Whiskers extend to show the range excluding outliers. Dots beyond the whiskers are outliers are unusually long durations.

#### 1. OSRM Time Distribution:

- Most values are tightly packed near 0–100 seconds.
- A few outliers exist, some between 500–1500 seconds.
- Indicates OSRM predictions are mostly conservative and don't deviate much.

#### 2. Actual Time Distribution:

- Still centered around a similar low median, but:
- Has many more and higher outliers, going up to 3000+ seconds.
- Shows greater variance than OSRM times.
- Suggests that in practice, delays or inefficiencies are frequent

### Conclusion

#### 1. OSRM Underestimation:

- The narrower box and fewer outliers suggest OSRM's model underestimates real-world variability — it gives more ideal-case estimates.

#### 2. Operational Delays:

- The wider spread in actual time points to unaccounted real-world factors such as:
  - Traffic congestion
  - Waiting times at loading/unloading
  - Road conditions or detours
  - Driver behavior or breaks

#### 3. Model Calibration Needed:

- There is a clear mismatch between predicted and real values.
- The routing model should be recalibrated or enhanced with real-time data or traffic history to be more reflective of reality.

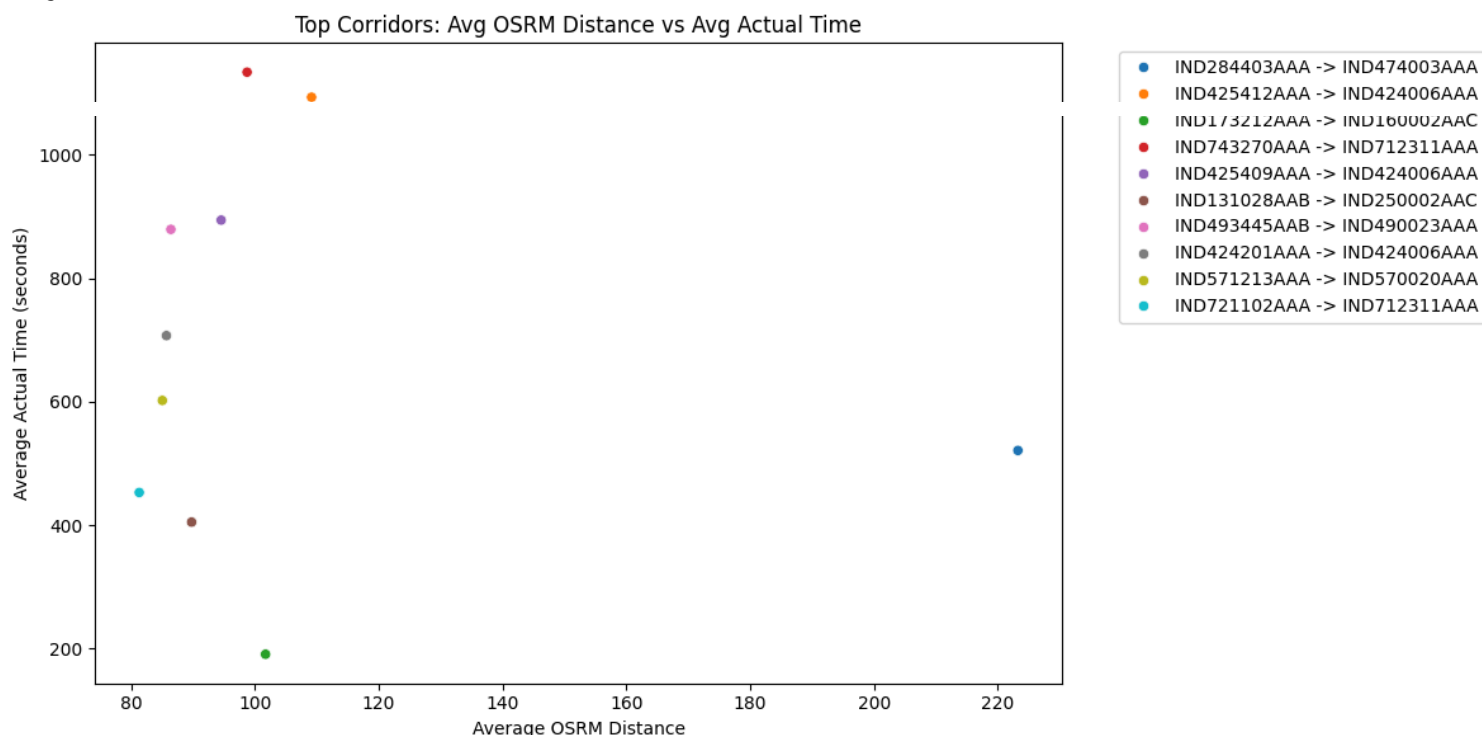
```
# Plot 4: Corridor-wise Avg Distance vs Time
corridor_stats = df.groupby('corridor').agg({
    'segment_osrm_distance': 'mean',
    'segment_actual_time': 'mean'
}).sort_values(by='segment_osrm_distance', ascending=False).head(10)
```

```
fig4, ax4 = plt.subplots(figsize=(12, 6))
sns.scatterplot(x='segment_osrm_distance', y='segment_actual_time', hue=corridor_stats.index, palette='tab10', data=corridor_stats, ax=ax4)
```

```
ax4.set_title("Top Corridors: Avg OSRM Distance vs Avg Actual Time")
ax4.set_xlabel("Average OSRM Distance")
ax4.set_ylabel("Average Actual Time (seconds)")
ax4.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

```
plt.tight_layout()
(fig1, fig2, fig3, fig4)
```

```
<Figure size 1000x600 with 1 Axes>,
<Figure size 1200x600 with 1 Axes>,
<Figure size 1000x600 with 1 Axes>,
<Figure size 1200x600 with 1 Axes>)
```



The scatter plot shows the relationship between average OSRM-predicted distance (likely in meters) and average actual time taken (in seconds), each point represents a corridor.

#### 1. General Trend:

- Normally, you'd expect a positive correlation (longer distance → more time), but here the data appears scattered.
- Some corridors have short OSRM distances but very high actual times, which breaks that expected trend.

#### 2. Outlier Corridors (High Time, Low Distance):

- Example: IND425412AAA -> IND424006AAA (orange dot) and IND173212AAA -> IND160002AAC (red dot)
  - Both have OSRM distances around 100–110 units but actual times >1100s.
  - Inefficient routes or delays — possibly high traffic congestion, bottlenecks, signal delays, etc.

#### 3. Efficient Corridor (Low Time, Decent Distance):

- Example: IND721102AAA -> IND712311AAA (green dot)
  - ~100 units of distance, but only ~200s actual time.
  - Very efficient travel, possibly due to clear roads or highway segments.

#### 4. Longest Distance:

- IND284403AAA -> IND474003AAA (dark blue dot)
  - ~220 OSRM units, actual time ~500s.
  - Despite being the longest route, its actual time is lower than many shorter ones, suggesting smooth flow or fast segments.

### Conclusion

#### 1. Distance ≠ Time Always:

- This graph makes it clear that routing distance alone isn't sufficient to predict actual travel time.
- Real-world conditions (traffic, road type, signals, construction, etc.) play a major role.

#### 2. Investigate Bottleneck Corridors:

- High-time, low-distance corridors like:
  - IND425412AAA -> IND424006AAA
  - IND173212AAA -> IND160002AAC

- Likely candidates for route optimization, traffic audits, or operational improvements.

### 3. Use This for ETA Improvements:

- Create a correction factor for corridors where time is disproportionately high.
- Consider building a regression model using distance, day/time, and location tags to predict actual travel time.

### 4. Cluster-Based Grouping:

- Cluster corridors by time/distance ratio to identify route classes:
  - Efficient (low time/distance)
  - Normal
  - Congested (high time/distance)

## Actionable Business Recommendations:

### 1. Optimize High-Volume Corridors

**Action:** Target the most frequently used delivery corridors to enhance efficiency and service delivery. This can be achieved through the following strategies:

- **Increase Delivery Slots:** Expand the number of available delivery windows to accommodate a higher volume of orders, ensuring customers have more opportunities to receive their shipments.
- **Allocate Larger Vehicle Fleets:** Deploy a greater number of delivery vehicles in these key corridors to streamline the delivery process, reduce wait times, and enhance overall fleet utilisation.
- **Reconfigure Hub Placement or Routes:** Analyse current logistics routes and hub locations to determine if adjustments can be made that would minimize travel distances and time, ultimately improving delivery speed and efficiency.

### 2. Improve Underperforming Corridors

**Action:** For corridors where actual delivery times significantly exceed the estimated times provided by the Open Source Routing Machine (OSRM), it's essential to identify and address specific issues:

- **Investigate Delays:** Conduct a thorough examination of factors contributing to delays, including identifying roadblocks, heavy traffic conditions, and challenges within warehousing processes that may impede timely deliveries.
- **Re-evaluate Logistics Partners or Delivery Models:** Assess the performance of current logistics partners and explore alternative delivery models to address inefficiencies. This may involve seeking out partners with proven track records in similar corridors or innovative delivery methodologies.

### 3. Rebalance Source Center Load

**Action:** When certain source centers are overwhelmingly responsible for the majority of shipments, it is important to redistribute the load:

- **Establish Satellite Hubs:** Create additional distribution hubs closer to high-demand regions to alleviate the burden on central hubs, thus improving response times and service to customers.
- **Reduce Delivery Pressure:** By spreading out the shipment load, reduce the risk of delays and bottlenecks associated with overloaded source centers, ensuring a smoother operation and faster delivery times.

### 4. Predictive Route Planning

**Action:** Leverage historical delivery data to enhance route planning and customer satisfaction:

- **Predict Delivery Time Ranges:** Use data analytics to develop accurate forecasts of delivery timeframes based on specific source-destination pairings, helping to inform customers more effectively about expected arrival times.
- **Build Accurate SLA Commitments:** Establish more precise Service Level Agreements (SLAs) based on real delivery performance data, ensuring commitments made to customers are both realistic and achievable.

### 5. Monitor and Benchmark Performance

**Action:** Develop a comprehensive set of Key Performance Indicators (KPIs) to evaluate and enhance delivery performance:

- **Measure % Deviation Between OSRM and Actual Time:** Track the differences between estimated and actual delivery times to identify areas for improvement and operational adjustments.
- **On-Time Delivery Rate per Corridor:** Calculate the percentage of deliveries that arrive within the promised time frame for each corridor, providing insight into performance consistency.
- **Analyze Top 10 Best/Worst Performing Routes:** Identify and analyze the top ten routes with the best and worst performance metrics, using this data to inform strategic decisions and implement targeted improvements.

---

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.