Business Case: Delhivery - Feature Engineering

About Delhivery

In the realm of modern commerce, efficient logistics operations play a pivotal role in ensuring the smooth flow of goods from source to destination. Delhivery, India's largest and fastest-growing integrated logistics player, stands at the forefront of this sector, leveraging cutting-edge technology and infrastructure to build the operating system for commerce. As part of their mission to continuously enhance operational quality, efficiency, and profitability, Delhivery relies on data-driven insights generated by its dedicated Data team.

The Data team at Delhivery is tasked with harnessing the vast volume of data generated by the company's logistics operations to derive actionable intelligence. This intelligence not only enables Delhivery to optimize its operations but also empowers it to stay ahead of competitors by offering superior service quality.

How can we help here?

The company wants to understand and process the data coming out of data engineering pipelines:

• Clean, sanitize and manipulate data to get useful features out of raw fields

• Make sense out of the raw data and help the data science team to build forecasting models on it

Let us start by importing required libraries

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
import math as m
from statsmodels.distributions.empirical_distribution import ECDF
from scipy.stats import ttest_ind
from scipy.stats import shapiro, kruskal, levene, expon, kstest
from scipy import stats
from statsmodels.graphics.gofplots import qqplot
from sklearn.preprocessing import MinMaxScaler, StandardScaler, LabelEncoder
import scipy.stats as spy
```

Loading the Dataset

In [2]:
```
df = pd.read_csv(r"C:\Users\hp\OneDrive\Desktop\delhivery_data.txt")
```

In [3]:
```
df.head()
```

Out[3]:

	data	trip_creation_time	route_schedule_uuid	route_type	trip_uuid	source_center	source_name	destination_center	destination_name	od_start_ti
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)	2018-09 03:21:32.418
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)	2018-09 03:21:32.418
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)	2018-09 03:21:32.418
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)	2018-09 03:21:32.418
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)	2018-09 03:21:32.418

5 rows × 24 columns

Columns in the Dataset

In [4]:
```
df.columns
```

Out[4]:
```
Index(['data', 'trip_creation_time', 'route_schedule_uuid', 'route_type',
       'trip_uuid', 'source_center', 'source_name', 'destination_center',
       'destination_name', 'od_start_time', 'od_end_time',
       'start_scan_to_end_scan', 'is_cutoff', 'cutoff_factor',
       'cutoff_timestamp', 'actual_distance_to_destination', 'actual_time',
       'osrm_time', 'osrm_distance', 'factor', 'segment_actual_time',
       'segment_osrm_time', 'segment_osrm_distance', 'segment_factor'],
      dtype='object')
```

Describing the Dataset

In [5]:
```
df.describe()
```

Out[5]:

	start_scan_to_end_scan	cutoff_factor	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	factor	segment_actual_time	segment_osrm_time
count	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.00000
mean	961.262986	232.926567	234.073372	416.927527	213.868272	284.771297	2.120107	36.196111	18.50754
std	1037.012769	344.755577	344.990009	598.103621	308.011085	421.119294	1.715421	53.571158	14.77596
min	20.000000	9.000000	9.000045	9.000000	6.000000	9.008200	0.144000	-244.000000	0.00000
25%	161.000000	22.000000	23.355874	51.000000	27.000000	29.914700	1.604264	20.000000	11.00000
50%	449.000000	66.000000	66.126571	132.000000	64.000000	78.525800	1.857143	29.000000	17.00000
75%	1634.000000	286.000000	286.708875	513.000000	257.000000	343.193250	2.213483	40.000000	22.00000
max	7898.000000	1927.000000	1927.447705	4532.000000	1686.000000	2326.199100	77.387097	3051.000000	1611.00000

Information about the Dataset

In [6]:
```
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
```

With the above information, we can see that our dataset has 144867 rows and 24 columns.

Conversion of categorical attributes to 'category'

In [7]:
```
df['data'] = df['data'].astype('category')
df['route_type'] = df['route_type'].astype('category')
```

Dropping the columns for which no information is given

In [8]:
```
df =df.drop(columns=['cutoff_factor','cutoff_timestamp','is_cutoff','factor','segment_factor'])
```

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144867 entries, 0 to 144866
Data columns (total 19 columns):
 #   Column                          Non-Null Count   Dtype   
---  ------                          --------------   -----   
 0   data                            144867 non-null  category
 1   trip_creation_time              144867 non-null  object  
 2   route_schedule_uuid             144867 non-null  object  
 3   route_type                      144867 non-null  category
 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  actual_distance_to_destination  144867 non-null  float64 
 13  actual_time                     144867 non-null  float64 
 14  osrm_time                       144867 non-null  float64 
 15  osrm_distance                   144867 non-null  float64 
 16  segment_actual_time             144867 non-null  float64 
 17  segment_osrm_time               144867 non-null  float64 
 18  segment_osrm_distance           144867 non-null  float64 
dtypes: category(2), float64(8), object(9)
memory usage: 19.1+ MB
```

Here we can notice that with dropping the unknown columns, the memory 25.6+ MB has been reduced to 21.0 + MB.

Updating the datatype of the datetime columns

In [10]:
```
df['trip_creation_time']=pd.to_datetime(df['trip_creation_time'])
df['od_start_time']=pd.to_datetime(df['od_start_time'])
df['od_end_time']=pd.to_datetime(df['od_end_time'])
```

Basic Data Cleaning and Exploration

Handle missing values in the data.

Checking the presence of NULL values in the Dataset

In [11]:
```
df.isnull().sum()
```

Out[11]:
```
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
actual_distance_to_destination      0
actual_time                         0
osrm_time                           0
osrm_distance                       0
segment_actual_time                 0
segment_osrm_time                   0
segment_osrm_distance               0
dtype: int64
```

Here we can see that there are different sum of NULL values present in source_name and destination_name.

Let us see the unique missing values in both source_name and destination_name by using their relative columns i.e. source_center and destination_center

In [12]:
```
# Filter rows where source_name & destination_name is missing and select unique combinations of source_center and  destination_center.
missing_source_names = df.loc[df['source_name'].isnull(), 'source_center'].unique()
missing_destination_names = df.loc[df['destination_name'].isnull(), 'destination_center'].unique()

# Create DataFrames for missing source names and destination names
missing_source_df = pd.DataFrame({'source_center': missing_source_names, 'missing_source_name': 'Yes'})
missing_destination_df = pd.DataFrame({'destination_center': missing_destination_names, 'missing_dest_name': 'Yes'})

# Merge the DataFrames based on common source_center and destination_center values
merged_df = pd.merge(missing_source_df, missing_destination_df, how='outer', left_on='source_center', right_on='destination_center')


# Display the result
print(merged_df)
```

```
   source_center missing_source_name destination_center missing_dest_name
0   IND342902A1B                 Yes       IND342902A1B               Yes
1   IND577116AAA                 Yes       IND577116AAA               Yes
2   IND282002AAD                 Yes       IND282002AAD               Yes
3   IND465333A1B                 Yes       IND465333A1B               Yes
4   IND841301AAC                 Yes       IND841301AAC               Yes
5   IND509103AAC                 Yes       IND509103AAC               Yes
6   IND126116AAA                 Yes       IND126116AAA               Yes
7   IND331022A1B                 Yes                NaN               NaN
8   IND505326AAB                 Yes       IND505326AAB               Yes
9   IND852118A1B                 Yes       IND852118A1B               Yes
10           NaN                 NaN       IND221005A1A               Yes
11           NaN                 NaN       IND250002AAC               Yes
12           NaN                 NaN       IND331001A1C               Yes
13           NaN                 NaN       IND122015AAC               Yes
```

Treating missing destination names and source names

In [13]:
```
num_missing_source_centers = len(missing_source_names)
num_missing_destination_centers = len(missing_destination_names)

source_labels = ['place_{}'.format(i) for i in range(1, num_missing_source_centers + 1)]
destination_labels = ['place_{}'.format(i) for i in range(1, num_missing_destination_centers + 1)]

for i, source_center in enumerate(missing_source_names):
    df.loc[(df['source_center'] == source_center) & (df['source_name'].isnull()), 'source_name'] = source_labels[i]

for i, destination_center in enumerate(missing_destination_names):
    df.loc[(df['destination_center'] == destination_center) & (df['destination_name'].isnull()), 'destination_name'] = destination_labels[i]
```

In [14]:
```
np.any(df.isnull())
```

Out[14]:
```
False
```

Analyze the structure of the data

Shape of the Dataset

In [15]:
```
df.shape
```

Out[15]:
```
(144867, 19)
```

Description of the Dataset

In [16]:
```
df.describe()
```

Out[16]:

	start_scan_to_end_scan	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	segment_actual_time	segment_osrm_time	segment_osrm_distance
count	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.000000	144867.00000
mean	961.262986	234.073372	416.927527	213.868272	284.771297	36.196111	18.507548	22.82902
std	1037.012769	344.990009	598.103621	308.011085	421.119294	53.571158	14.775960	17.86066
min	20.000000	9.000045	9.000000	6.000000	9.008200	-244.000000	0.000000	0.00000
25%	161.000000	23.355874	51.000000	27.000000	29.914700	20.000000	11.000000	12.07010
50%	449.000000	66.126571	132.000000	64.000000	78.525800	29.000000	17.000000	23.51300
75%	1634.000000	286.708875	513.000000	257.000000	343.193250	40.000000	22.000000	27.81325
max	7898.000000	1927.447705	4532.000000	1686.000000	2326.199100	3051.000000	1611.000000	2191.40370

Type of data in the Dataset

In [17]:

```
df.dtypes
```

Out[17]:

```
data                                    category
trip_creation_time                datetime64[ns]
route_schedule_uuid                       object
route_type                              category
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
actual_distance_to_destination           float64
actual_time                              float64
osrm_time                                float64
osrm_distance                            float64
segment_actual_time                      float64
segment_osrm_time                        float64
segment_osrm_distance                    float64
dtype: object
```

Merging the rows

When analyzing delivery details, merging multiple rows representing a single package's journey is key. Aggregation methods condense this data while preserving accuracy. Numeric fields transform, providing insights into overall performance. Choosing the right aggregation methods is crucial for effective analysis.

In [18]:

```
dict={
    'data': 'first',
    'route_type': 'first',
    'trip_creation_time': 'first',
    'source_name': 'first',
    'destination_name': 'last',
    'od_start_time': 'first',
    'od_end_time': 'first',
    'start_scan_to_end_scan': 'first',
    'actual_distance_to_destination': 'last',
    'actual_time': 'last',
    'osrm_time': 'last',
    'osrm_distance': 'last',
    'segment_actual_time': 'sum',
    'segment_osrm_time': 'sum',
    'segment_osrm_distance': 'sum'
}

gdf = df.groupby(by=['trip_uuid', 'source_center', 'destination_center'], as_index=False).agg(dict)
gdf
```

Out[18]:

	trip_uuid	source_center	destination_center	data	route_type	trip_creation_time	source_name	destination_name	od_start_time	od_enc
0	trip-153671041653548748	IND209304AAA	IND000000ACB	training	FTL	2018-09-12 00:00:16.535741	Kanpur_Central_H_6 (Uttar Pradesh)	Gurgaon_Bilaspur_HB (Haryana)	2018-09-12 16:39:46.858469	2018 13:40:23.1
1	trip-153671041653548748	IND462022AAA	IND209304AAA	training	FTL	2018-09-12 00:00:16.535741	Bhopal_Trnsport_H (Madhya Pradesh)	Kanpur_Central_H_6 (Uttar Pradesh)	2018-09-12 00:00:16.535741	2018 16:39:46.8
2	trip-153671042288605164	IND561203AAB	IND562101AAA	training	Carting	2018-09-12 00:00:22.886430	Doddablpur_ChikaDPP_D (Karnataka)	Chikblapur_ShntiSgr_D (Karnataka)	2018-09-12 02:03:09.655591	2018 03:01:59.5
3	trip-153671042288605164	IND572101AAA	IND561203AAB	training	Carting	2018-09-12 00:00:22.886430	Tumkur_Veersagr_I (Karnataka)	Doddablpur_ChikaDPP_D (Karnataka)	2018-09-12 00:00:22.886430	2018 02:03:09.6
4	trip-153671043369099517	IND000000ACB	IND160002AAC	training	FTL	2018-09-12 00:00:33.691250	Gurgaon_Bilaspur_HB (Haryana)	Chandigarh_Mehmdpur_H (Punjab)	2018-09-14 03:40:17.106733	2018 17:34:55.4
...	...	...	...	...	...	...	...	...	...	
26363	trip-153861115439069069	IND628204AAA	IND627657AAA	test	Carting	2018-10-03 23:59:14.390954	Tirchchndr_Shnmgprm_D (Tamil Nadu)	Thisayanvilai_UdnkdiRD_D (Tamil Nadu)	2018-10-04 02:29:04.272194	2018 03:31:11.1
26364	trip-153861115439069069	IND628613AAA	IND627005AAA	test	Carting	2018-10-03 23:59:14.390954	Peikulam_SriVnktpm_D (Tamil Nadu)	Tirunelveli_VdkkuSrt_I (Tamil Nadu)	2018-10-04 04:16:39.894872	2018 05:47:45.1
26365	trip-153861115439069069	IND628801AAA	IND628204AAA	test	Carting	2018-10-03 23:59:14.390954	Eral_Busstand_D (Tamil Nadu)	Tirchchndr_Shnmgprm_D (Tamil Nadu)	2018-10-04 01:44:53.808000	2018 02:29:04.2
26366	trip-153861118270144424	IND583119AAA	IND583101AAA	test	FTL	2018-10-03 23:59:42.701692	Sandur_WrdN1DPP_D (Karnataka)	Bellary_Dc (Karnataka)	2018-10-04 03:58:40.726547	2018 08:46:09.1
26367	trip-153861118270144424	IND583201AAA	IND583119AAA	test	FTL	2018-10-03 23:59:42.701692	Hospet (Karnataka)	Sandur_WrdN1DPP_D (Karnataka)	2018-10-04 02:51:44.712656	2018 03:58:40.7

26368 rows × 18 columns

Build some features to prepare the data for actual analysis. Extract features from the below fields.

Destination Name: Split & extract features out of destination. City-place-code (State)

In [19]:
```
def location_to_state(a):
    l = a.split('(')
    if len(l) == 1:
        return l[0]
    else:
        return l[1].replace(')', "")
```

In [20]:
```
def location_to_city(a):
    if 'location' in a:
        return 'unknown_city'
    else:
        l = a.split()[0].split('_')
        if 'CCU' in a:
            return 'Kolkata'
        elif ('HBR' in a.upper()) or ('BLR' in a.upper()):
            return 'Bengaluru'
        elif 'MAA' in a.upper():
            return 'Chennai'
        elif 'FBD' in a.upper():
            return 'Faridabad'
        elif 'DEL' in a.upper():
            return 'Delhi'
        elif 'BOM' in a.upper():
            return 'Mumbai'
        elif 'OK' in a.upper():
            return 'Delhi'
        elif 'GZB' in a.upper():
            return 'Ghaziabad'
        elif 'HYD' in a.upper():
            return 'Hyderabad'
        elif 'GGN' in a.upper():
            return 'Gurgaon'
        elif 'AMD' in a.upper():
            return 'Ahmedabad'
        elif 'CJB' in a.upper():
            return 'Coimbatore'
        return l[0]
```

In [21]:
```
def location_to_place(a):
    if 'location' in a:
        return a
    elif 'HBR' in a:
        return 'HBR Layout PC'
    else:
        l = a.split()[0].split('_', 1)
        if len(l) == 1:
            return 'unknown_place'
        else:
            return l[1]
```

In [22]:
```
# for destination state
gdf['destination_state'] = gdf['destination_name'].apply(location_to_state)
print('No of destination state :', gdf['destination_state'].nunique())

# for destination city
gdf['destination_city'] = gdf['destination_name'].apply(location_to_city)
print('No of destination city :', gdf['destination_city'].nunique())

# for destination place
gdf['destination_place'] = gdf['destination_name'].apply(location_to_place)
print('No of destination place :', gdf['destination_place'].nunique())


gdf[['destination_state','destination_city','destination_place']].head()
```

```
No of destination state : 45
No of destination city : 1197
No of destination place : 1191
```

Out[22]:

	destination_state	destination_city	destination_place
0	Haryana	Gurgaon	Bilaspur_HB
1	Uttar Pradesh	Kanpur	Central_H_6
2	Karnataka	Chikblapur	ShntiSgr_D
3	Karnataka	Doddablpur	ChikaDPP_D
4	Punjab	Chandigarh	Mehmdpur_H

Source Name: Split and extract features out of destination. City-place-code (State)

In [23]:
```
# for source state
gdf['source_state'] = gdf['destination_name'].apply(location_to_state)
print('No of source state :', gdf['source_state'].nunique())

# for source city
gdf['source_city'] = gdf['source_name'].apply(location_to_city)
print('No of source_city :', gdf['source_city'].nunique())

# for source place
gdf['source_place'] = gdf['source_name'].apply(location_to_place)
print('No of source_place :', gdf['source_place'].nunique())


gdf[['source_state','source_city','source_place']].head()
```

```
No of source state : 45
No of source_city : 1203
No of source_place : 1217
```

Out[23]:

	source_state	source_city	source_place
0	Haryana	Kanpur	Central_H_6
1	Uttar Pradesh	Bhopal	Trnsport_H
2	Karnataka	Doddablpur	ChikaDPP_D
3	Karnataka	Tumkur	Veersagr_I
4	Punjab	Gurgaon	Bilaspur_HB

Trip_creation_time: Extract features like month, year and day etc

In [24]:

```
gdf['trip_creation_date'] = pd.to_datetime(gdf['trip_creation_time'].dt.date)
gdf['trip_creation_date'].head()
```

Out[24]:

```
0    2018-09-12
1    2018-09-12
2    2018-09-12
3    2018-09-12
4    2018-09-12
Name: trip_creation_date, dtype: datetime64[ns]
```

In [25]:

```
# For day of the month
gdf['trip_creation_day'] = gdf['trip_creation_time'].dt.day
gdf['trip_creation_day'] = gdf['trip_creation_day'].astype('int8')

# For month
gdf['trip_creation_month'] = gdf['trip_creation_time'].dt.month
gdf['trip_creation_month'] = gdf['trip_creation_month'].astype('int8')

# For year
gdf['trip_creation_year'] = gdf['trip_creation_time'].dt.year
gdf['trip_creation_year'] = gdf['trip_creation_year'].astype('int16')

# For week
gdf['trip_creation_week'] = gdf['trip_creation_time'].dt.isocalendar().week
gdf['trip_creation_week'] = gdf['trip_creation_week'].astype('int8')

# For hour
gdf['trip_creation_hour'] = gdf['trip_creation_time'].dt.hour
gdf['trip_creation_hour'] = gdf['trip_creation_hour'].astype('int8')

# For day of the week
gdf['trip_creation_day_name'] = gdf['trip_creation_time'].dt.day_name()

gdf[['trip_creation_day' , 'trip_creation_month', 'trip_creation_year' ,'trip_creation_week', 'trip_creation_hour','trip_creation_day_name']].head()
```

Out[25]:

	trip_creation_day	trip_creation_month	trip_creation_year	trip_creation_week	trip_creation_hour	trip_creation_day_name
0	12	9	2018	37	0	Wednesday
1	12	9	2018	37	0	Wednesday
2	12	9	2018	37	0	Wednesday
3	12	9	2018	37	0	Wednesday
4	12	9	2018	37	0	Wednesday

In [26]:

```
gdf.shape
```

Out[26]:

```
(26368, 31)
```

3. In-depth analysis and feature engineering:

Calculate the time taken between od_start_time and od_end_time and keep it as a feature. Drop the original columns, if required

In [27]:

```
gdf['od_total_time'] = gdf['od_end_time'] - gdf['od_start_time']
gdf.drop(columns = ['od_end_time', 'od_start_time'], inplace = True)
gdf['od_total_time'] = gdf['od_total_time'].apply(lambda x : round(x.total_seconds() / 60.0, 2))
```

In [28]:

```
gdf.head(3)
```

Out[28]:

	trip_uuid	source_center	destination_center	data	route_type	trip_creation_time	source_name	destination_name	start_scan_to_end_scan	actual_dis
0	trip-153671041653548748	IND209304AAA	IND000000ACB	training	FTL	2018-09-12 00:00:16.535741	Kanpur_Central_H_6 (Uttar Pradesh)	Gurgaon_Bilaspur_HB (Haryana)	1260.0	
1	trip-153671041653548748	IND462022AAA	IND209304AAA	training	FTL	2018-09-12 00:00:16.535741	Bhopal_Trnsport_H (Madhya Pradesh)	Kanpur_Central_H_6 (Uttar Pradesh)	999.0	
2	trip-153671042288605164	IND561203AAB	IND562101AAA	training	Carting	2018-09-12 00:00:22.886430	Doddablpur_ChikaDPP_D (Karnataka)	Chikblapur_ShntiSgr_D (Karnataka)	58.0	

3 rows × 30 columns

In [29]:

```
dict_2 = {
    'source_center' : 'first',
    'destination_center' : 'last',
    'data' : 'first',
    'route_type' : 'first',
    'trip_creation_time' : 'first',
    'source_name' : 'first',
    'destination_name' : 'last',
```

```
    'od_total_time': 'first',
    'start_scan_to_end_scan': 'sum',
    'actual_distance_to_destination': 'sum',
    'actual_time': 'sum',
    'osrm_time': 'sum',
    'osrm_distance': 'sum',
    'segment_actual_time': 'sum',
    'segment_osrm_time': 'sum',
    'segment_osrm_distance': 'sum',
    'source_state': 'first',
    'source_city': 'first',
    'source_place': 'first',
    'destination_city': 'last',
    'destination_place': 'last',
    'destination_state': 'last',
    'trip_creation_day': 'first',
    'trip_creation_month': 'first',
    'trip_creation_year': 'first',
    'trip_creation_week': 'first',
    'trip_creation_hour': 'first',
    'trip_creation_day_name': 'first'
}

gdf_2 = gdf.groupby(by='trip_uuid', as_index=False).agg(dict_2)
gdf_2.head()
```

Out[29]:

	trip_uuid	source_center	destination_center	data	route_type	trip_creation_time	source_name	destination_name	od_total_time	start_scan_to_en
0	trip-153671041653548748	IND209304AAA	IND209304AAA	training	FTL	2018-09-12 00:00:16.535741	Kanpur_Central_H_6 (Uttar Pradesh)	Kanpur_Central_H_6 (Uttar Pradesh)	1260.60	
1	trip-153671042288605164	IND561203AAB	IND561203AAB	training	Carting	2018-09-12 00:00:22.886430	Doddablpur_ChikaDPP_D (Karnataka)	Doddablpur_ChikaDPP_D (Karnataka)	58.83	
2	trip-153671043369099517	IND000000ACB	IND000000ACB	training	FTL	2018-09-12 00:00:33.691250	Gurgaon_Bilaspur_HB (Haryana)	Gurgaon_Bilaspur_HB (Haryana)	834.64	
3	trip-153671046011330457	IND400072AAB	IND401104AAA	training	Carting	2018-09-12 00:01:00.113710	Mumbai Hub (Maharashtra)	Mumbai_MiraRd_IP (Maharashtra)	100.49	
4	trip-153671052974046625	IND583101AAA	IND583119AAA	training	FTL	2018-09-12 00:02:09.740725	Bellary_Dc (Karnataka)	Sandur_WrdN1DPP_D (Karnataka)	152.01	

5 rows × 29 columns

Hypothesis testing using T-test

STEP-1: Establish the Null Hypothesis

Null Hypothesis (H0) - There is no significant difference between X variable & Y variable.

Alternate Hypothesis (HA) - There is a significant difference between X variable & Y variable.

STEP-2 : Validate Assumptions for the Hypothesis

Check the distribution using a QQ Plot. Assess the homogeneity of variances using Levene's test.

STEP-3 Determine Test Statistics and Distribution under H0.

If assumptions for t-test are satisfied, we will move forward with the t-test for independent samples. Otherwise, perform a non-parametric test equivalent to the t-test for independent samples, such as the Mann-Whitney U rank test.

STEP-4 Calculate p-value & Set Alpha

Set significance level (alpha) to 0.05

STEP-5 Compare p-value & alpha.

p-val > alpha** : Accept H0

p-val < alpha** : Fail to reject H0

Compare the difference between od_total_time and start_scan_to_end_scan. Do hypothesis testing/ Visual analysis to check.

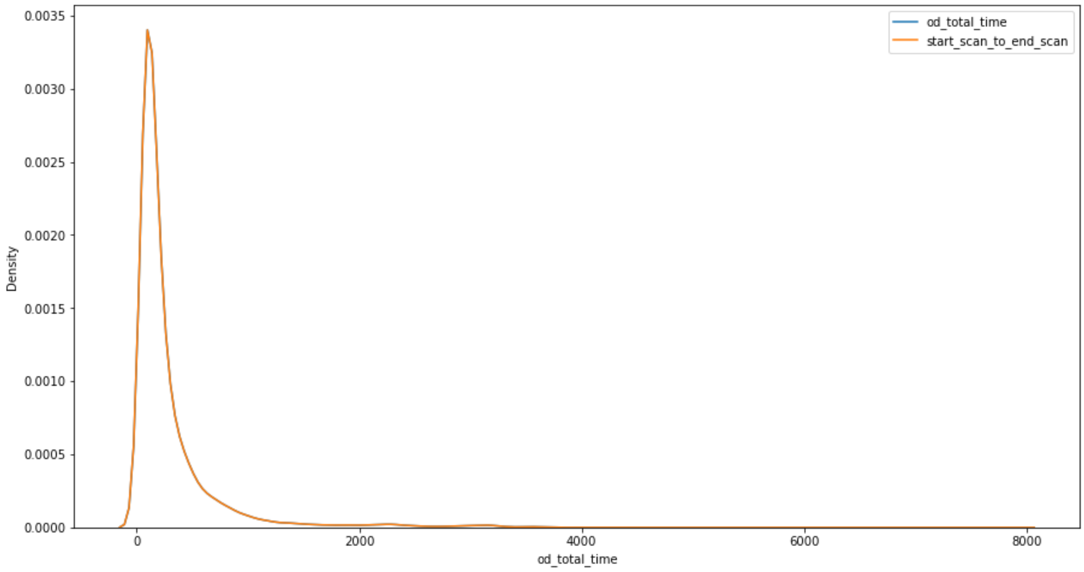
In [30]:
```
gdf_2[['od_total_time', 'start_scan_to_end_scan']].describe()
```

Out[30]:

	od_total_time	start_scan_to_end_scan
count	14817.000000	14817.000000
mean	340.508373	530.810016
std	505.656648	658.705957
min	23.000000	23.000000
25%	104.160000	149.000000
50%	175.030000	280.000000
75%	334.750000	637.000000
max	7898.550000	7898.000000

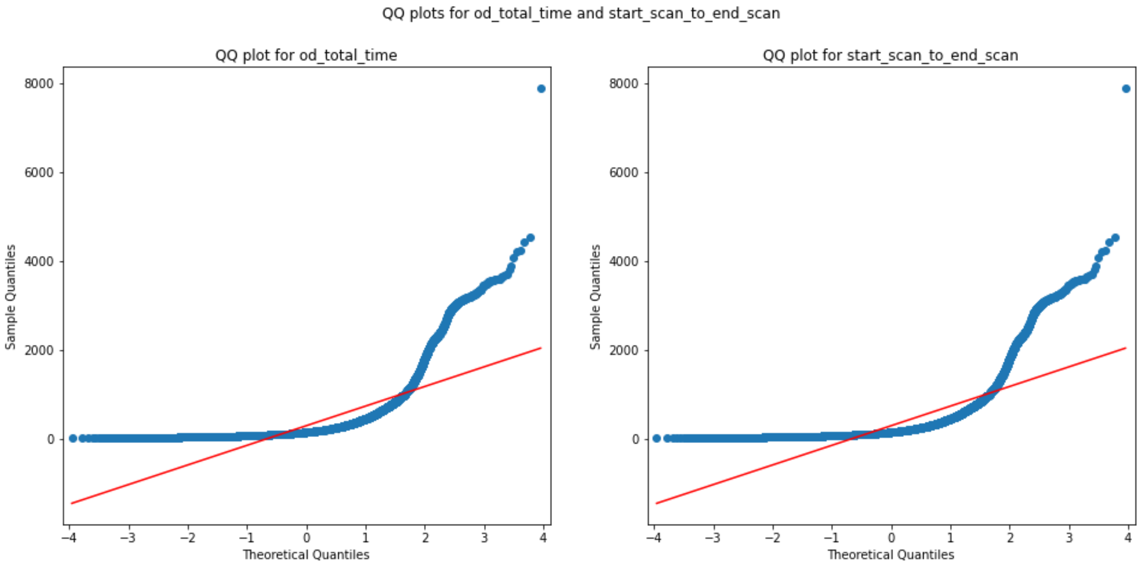
Visual Checks to Assess Normality of Samples

In [31]:
```
plt.figure(figsize=(15, 8))  # Adjust width and height as needed
sns.kdeplot(gdf["od_total_time"], label="od_total_time")
sns.kdeplot(gdf["start_scan_to_end_scan"], label="start_scan_to_end_scan")
plt.legend()
plt.show()
```



QQ Plot for Normality

In [32]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
qqplot(gdf["od_total_time"], line='s', ax=ax1)
ax1.set_title('QQ plot for od_total_time')
qqplot(gdf["start_scan_to_end_scan"], line='s', ax=ax2)
ax2.set_title('QQ plot for start_scan_to_end_scan')
plt.suptitle('QQ plots for od_total_time and start_scan_to_end_scan')
plt.show()
```



- The plots above indicate that the samples are not drawn from a normal distribution.

Shapiro-Wilk test for normality

H_0 : The sample exhibits normal distribution

H_1 : The sample does not exhibit normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

In [33]:
```
test_stat, p_value = shapiro(gdf_2['od_total_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

In [34]:
```
test_stat, p_value = shapiro(gdf_2['start_scan_to_end_scan'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

Lavene's test for Homogeneity of Variances

In [35]:
```
# Null Hypothesis(H0) - Variance is Homogenous

# Alternate Hypothesis(HA) - Variance is Non Homogenous

test_stat, p_value = levene(gdf_2['od_total_time'], gdf_2['start_scan_to_end_scan'])
print('p-value : ', p_value)

if p_value < 0.05:
    print('Samples do not have Homogeneity in Variance')
else:
    print('Samples have Homogeneity in Variance ')
```

```
p-value :  1.09817016976443666e-112
Samples do not have Homogeneity in Variance
```

As the assumptions of T-test are not satisfied, we are performing the non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

In [36]:
```
Test_stat, p_value = spy.mannwhitneyu(gdf_2['od_total_time'], gdf_2['start_scan_to_end_scan'])
print('p-value :', p_value)
if p_value < 0.05:
 print('The samples are not similar')
else:
 print('The samples are similar ')
```

```
p-value : 0.0
The samples are not similar
```

Conclusion - The "od_total_time" and "start_scan_to_end_scan" are not similar, as p-value < alpha.

Compare difference between actual_time aggregated value and OSRM time aggregated value (aggregated values are the values you'll get after merging the rows on the basis of trip_uuid). Do hypothesis testing / visual analysis to check.

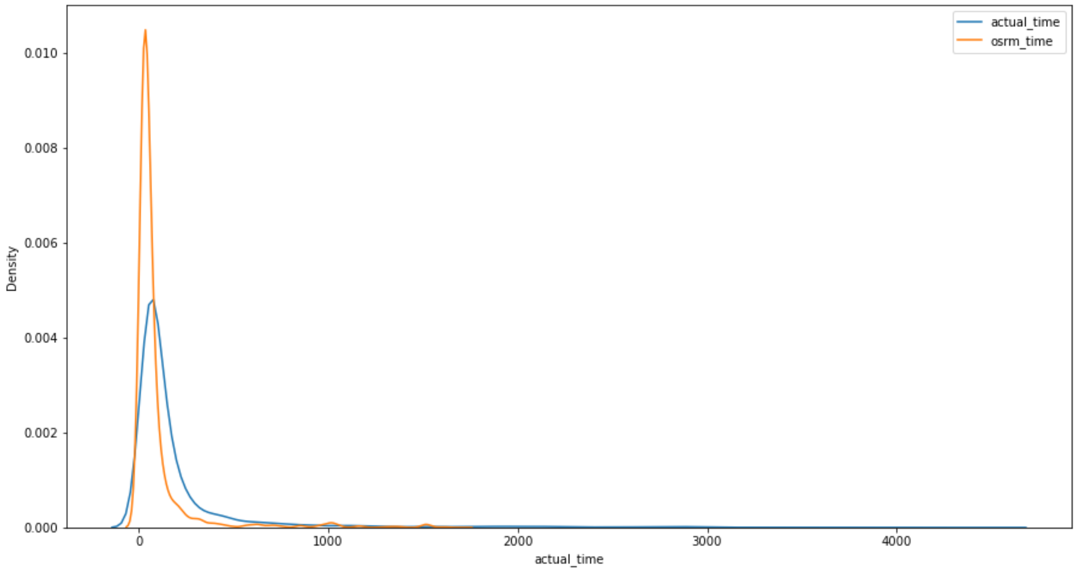
In [37]:
```
gdf_2[['actual_time', 'osrm_time']].describe()
```

Out[37]:

	actual_time	osrm_time
count	14817.000000	14817.000000
mean	357.143754	161.384018
std	561.396157	271.360995
min	9.000000	6.000000
25%	67.000000	29.000000
50%	149.000000	60.000000
75%	370.000000	168.000000
max	6265.000000	2032.000000

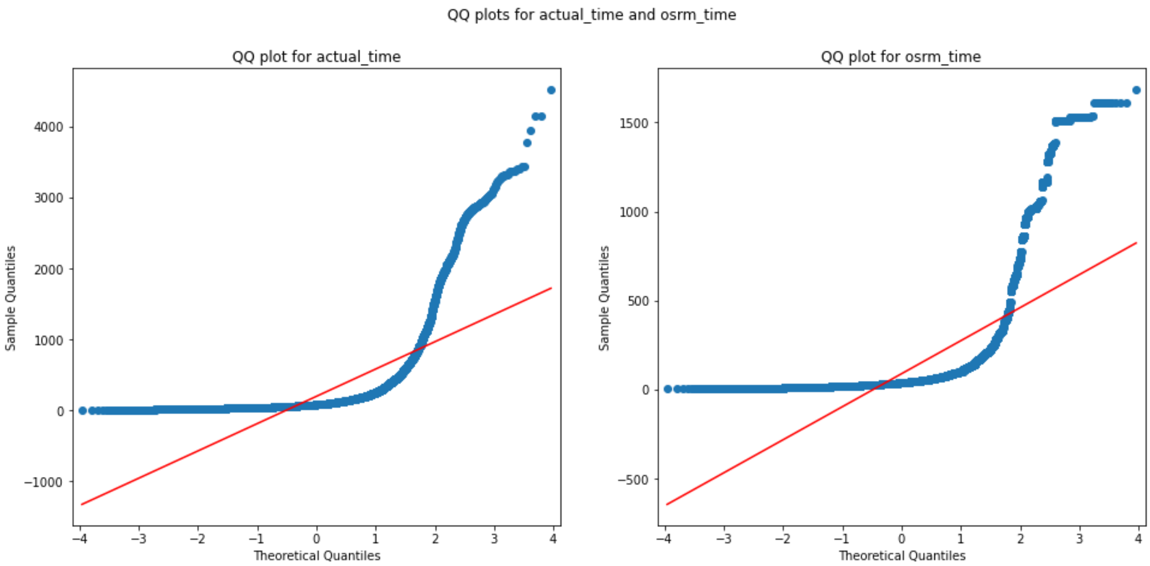
Visual Checks to Assess Normality of Samples

In [38]:
```
plt.figure(figsize=(15, 8))  # Adjust width and height as needed
sns.kdeplot(gdf["actual_time"], label="actual_time")
sns.kdeplot(gdf["osrm_time"], label="osrm_time")
plt.legend()
plt.show()
```



In [39]:
```
#### QQ Plot for Normality
```

In [40]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
qqplot(gdf["actual_time"], line='s', ax=ax1)
ax1.set_title('QQ plot for actual_time')
qqplot(gdf["osrm_time"], line='s', ax=ax2)
ax2.set_title('QQ plot for osrm_time')
plt.suptitle('QQ plots for actual_time and osrm_time')
plt.show()
```



- The plots above indicate that the samples are not drawn from a normal distribution.

Shapiro-Wilk test for normality

H_0 : The sample exhibits normal distribution

H_1 : The sample does not exhibit normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

In [41]:
```
test_stat, p_value = shapiro(gdf_2['actual_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

In [42]:
```
test_stat, p_value = shapiro(gdf_2['osrm_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

Lavene's test for Homogeneity of Variances

In [43]:
```
# Null Hypothesis(H0) - Variance is Homogenous

# Alternate Hypothesis(HA) - Variance is Non Homogenous

test_stat, p_value = levene(gdf_2['actual_time'], gdf_2['osrm_time'])
print('p-value : ', p_value)

if p_value < 0.05:
    print('Samples do not have Homogeneity in Variance')
else:
    print('Samples have Homogeneity in Variance ')
```

```
p-value :  1.8712979936832 08e-220
Samples do not have Homogeneity in Variance
```

As the assumptions of T-test are not satisfied, we are performing the non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

In [44]:
```
Test_stat, p_value = spy.mannwhitneyu(gdf_2['actual_time'], gdf_2['osrm_time'])
print('p-value :', p_value)
if p_value < 0.05:
 print('The samples are not similar')
else:
 print('The samples are similar ')
```

```
p-value : 0.0
The samples are not similar
```

Conclusion - The "actual_time" and "osrm_time" are not similar, as p-value < alpha.

Compare difference between actual_time aggregated value and segment actual time aggregated value. Do hypothesis testing / visual analysis to check.

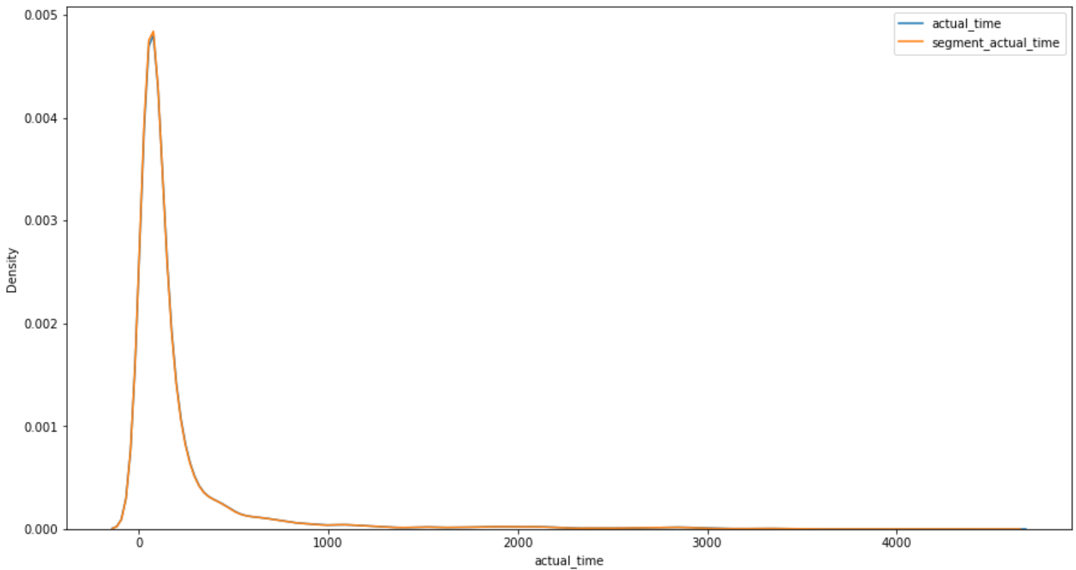
In [45]:
```
gdf_2[['actual_time', 'segment_actual_time']].describe()
```

Out[45]:

	actual_time	segment_actual_time
count	14817.000000	14817.000000
mean	357.143754	353.892286
std	561.396157	556.247965
min	9.000000	9.000000
25%	67.000000	66.000000
50%	149.000000	147.000000
75%	370.000000	367.000000
max	6265.000000	6230.000000

Visual Checks to Assess Normality of Samples

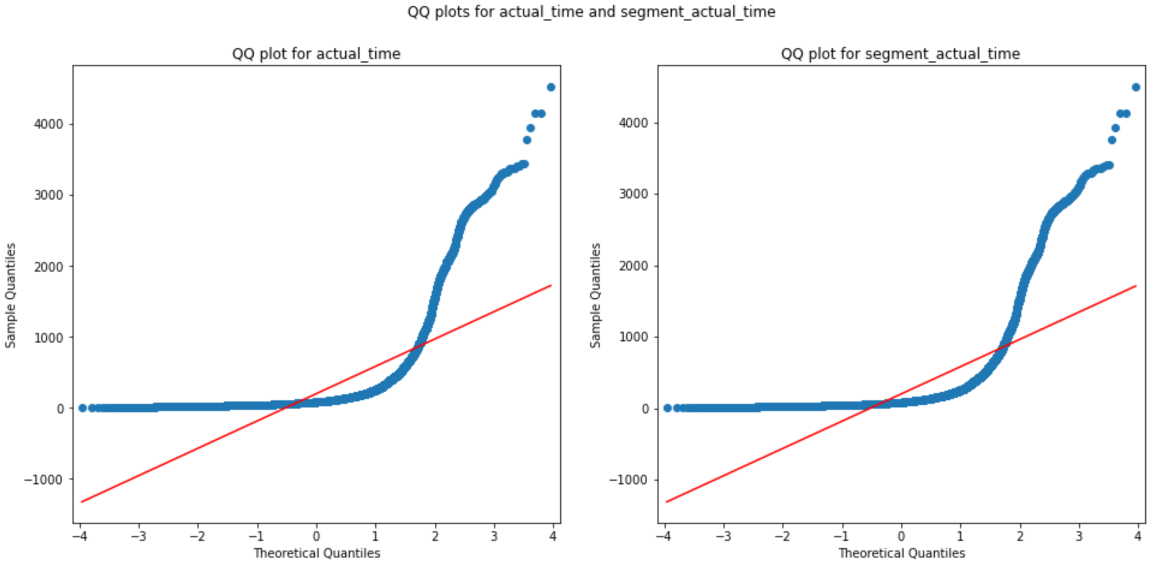
In [46]:
```
plt.figure(figsize=(15, 8))  # Adjust width and height as needed
sns.kdeplot(gdf["actual_time"], label='actual_time'  )
sns.kdeplot(gdf["segment_actual_time"], label= 'segment_actual_time' )
plt.legend()
plt.show()
```



QQ Plot for Normality

In [47]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
qqplot(gdf["actual_time"], line='s', ax=ax1)
ax1.set_title('QQ plot for actual_time')
qqplot(gdf["segment_actual_time"], line='s', ax=ax2)
ax2.set_title('QQ plot for segment_actual_time')
```

```
plt.suptitle('QQ plots for actual_time and segment_actual_time')
plt.show()
```



- The plots above indicate that the samples are not drawn from a normal distribution.

Shapiro-Wilk test for normality

H_0 : The sample exhibits normal distribution

H_1 : The sample does not exhibit normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

In [48]:
```
test_stat, p_value = spy.shapiro(gdf_2['actual_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

In [49]:
```
test_stat, p_value = spy.shapiro(gdf_2['segment_actual_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

Lavene's test for Homogeneity of Variances

In [50]:
```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(gdf_2['actual_time'], gdf_2['segment_actual_time'])
print('p-value : ', p_value)

if p_value < 0.05:
    print('The samples do not have Homogenous Variance')
else:
    print('The samples have Homogenous Variance ')
```

```
p-value :  0.6955022668700895
The samples have Homogenous Variance
```

As the normality assumption of T-test is not satisfied, we are performing the non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

In [51]:
```
Test_stat, p_value = spy.mannwhitneyu(gdf_2['actual_time'], gdf_2['segment_actual_time'])
print('p-value :', p_value)
if p_value < 0.05:
 print('The samples are not similar')
else:
 print('The samples are similar ')
```

```
p-value : 0.4164235159622476
The samples are similar
```

Conclusion - The "actual_time" and "segment_actual_time" are similar, as p-value > alpha.

Compare difference between osrm distance aggregated value and segment osrm distance aggregated value. Do hypothesis testing / visual analysis to check.

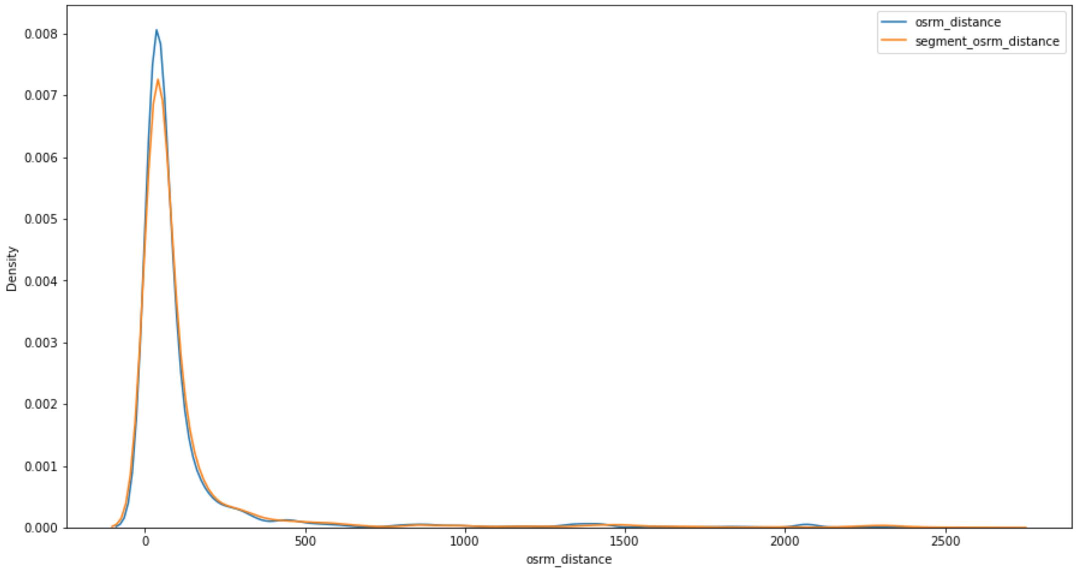
In [52]:
```
gdf_2[['osrm_distance', 'segment_osrm_distance']].describe()
```

Out[52]:

	osrm_distance	segment_osrm_distance
count	14817.000000	14817.000000
mean	204.344689	223.201161
std	370.395573	416.628374
min	9.072900	9.072900
25%	30.819200	32.654500
50%	65.618800	70.154400
75%	208.475000	218.802400
max	2840.081000	3523.632400

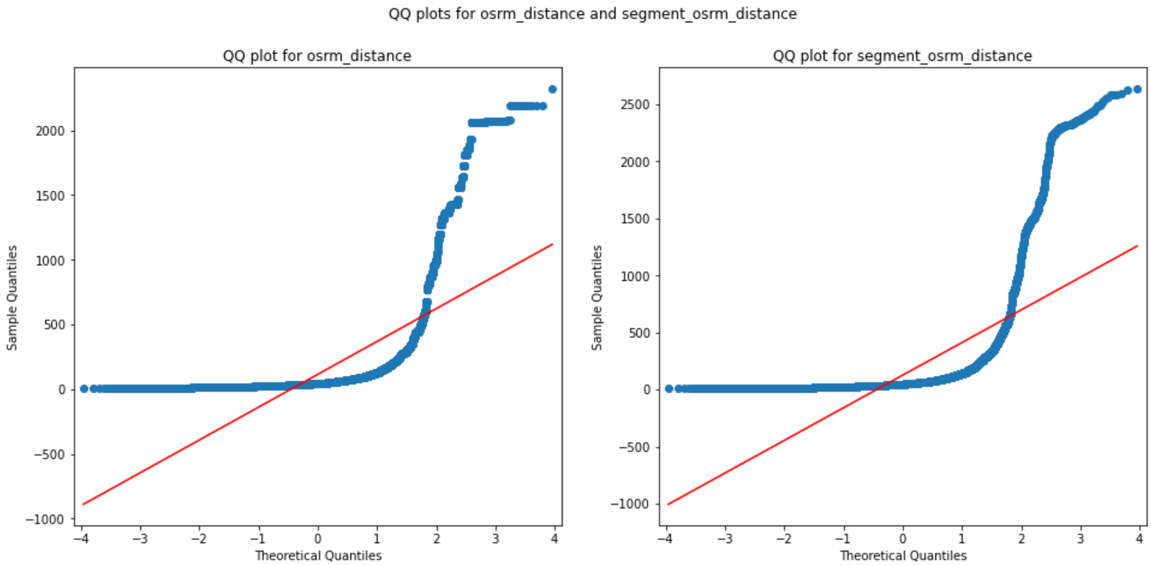
Visual Checks to Assess Normality of Samples

In [53]:
```
plt.figure(figsize=(15, 8))  # Adjust width and height as needed
sns.kdeplot(gdf["osrm_distance"], label='osrm_distance' )
sns.kdeplot(gdf["segment_osrm_distance"], label='segment_osrm_distance' )
plt.legend()
plt.show()
```



QQ Plot for Normality

In [54]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
qqplot(gdf["osrm_distance"], line='s', ax=ax1)
ax1.set_title('QQ plot for osrm_distance')
qqplot(gdf["segment_osrm_distance"], line='s', ax=ax2)
ax2.set_title('QQ plot for segment_osrm_distance')
plt.suptitle('QQ plots for osrm_distance and segment_osrm_distance')
plt.show()
```



- The plots above indicate that the samples are not drawn from a normal distribution.

Shapiro-Wilk test for normality

H_0 : The sample exhibits normal distribution

H_1 : The sample does not exhibit normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

In [55]:
```
test_stat, p_value = shapiro(gdf_2['osrm_distance'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

In [56]:
```
test_stat, p_value = shapiro(gdf_2['segment_osrm_distance'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

Lavene's test for Homogeneity of Variances

In [57]:
```
# Null Hypothesis(H0) - Variance is Homogenous

# Alternate Hypothesis(HA) - Variance is Non Homogenous

test_stat, p_value = levene(gdf_2['osrm_distance'], gdf_2['segment_osrm_distance'])
print('p-value : ', p_value)

if p_value < 0.05:
    print('Samples do not have Homogeneity in Variance')
else:
    print('Samples have Homogeneity in Variance ')
```

```
p-value :  0.00020976354422600578
Samples do not have Homogeneity in Variance
```

As the assumptions of T-test are not satisfied, we are performing the non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

In [58]:
```
Test_stat, p_value = spy.mannwhitneyu(gdf_2['osrm_distance'], gdf_2['segment_osrm_distance'])
print('p-value :', p_value)
if p_value < 0.05:
 print('The samples are not similar')
else:
 print('The samples are similar ')
```

```
p-value : 9.5113835882763 73e-07
The samples are not similar
```

Conclusion - The "osrm_distance" and "segment_osrm_distance" are not similar, as p-value < alpha.

Conduct hypothesis testing and visual analysis comparing the aggregated values of "osrm_time" and "segment_osrm_time" obtained after merging rows based on the trip_uuid column.

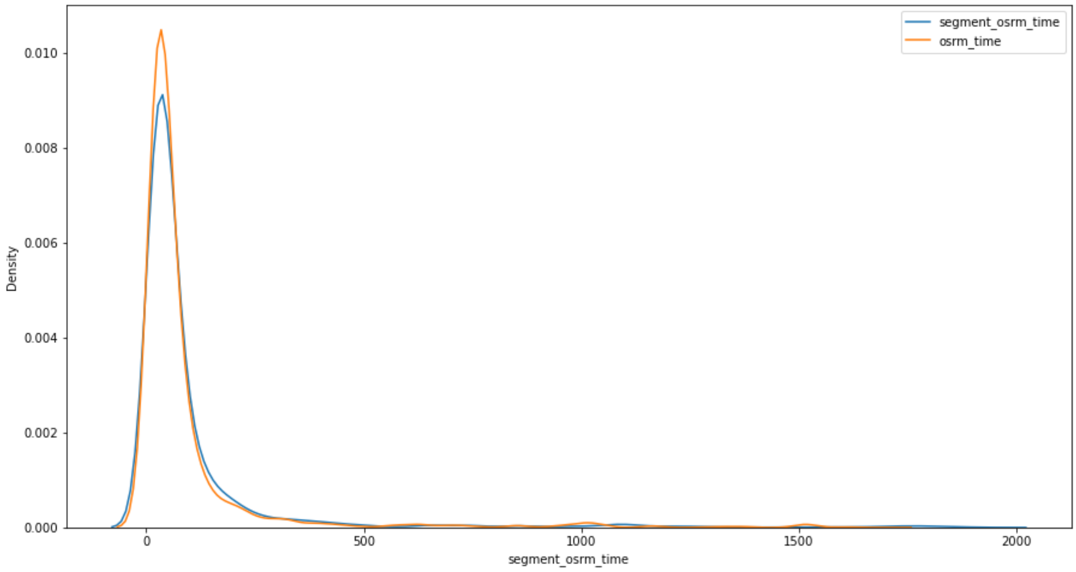
In [59]:
```
gdf_2[['osrm_time', 'segment_osrm_time']].describe()
```

Out[59]:

	osrm_time	segment_osrm_time
count	14817.000000	14817.000000
mean	161.384018	180.949787
std	271.360995	314.542047
min	6.000000	6.000000
25%	29.000000	31.000000
50%	60.000000	65.000000
75%	168.000000	185.000000
max	2032.000000	2564.000000

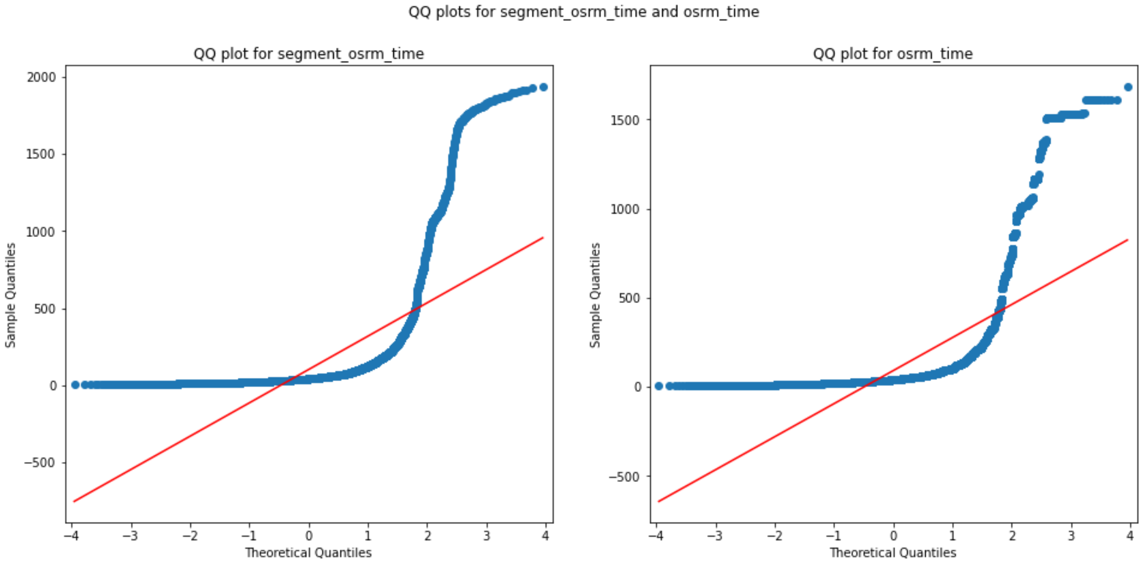
Visual Checks to Assess Normality of Samples

In [60]:
```
plt.figure(figsize=(15, 8))  # Adjust width and height as needed
sns.kdeplot(gdf["segment_osrm_time"], label="segment_osrm_time")
sns.kdeplot(gdf["osrm_time"], label="osrm_time")
plt.legend()
plt.show()
```



QQ Plot for Normality

In [61]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
qqplot(gdf["segment_osrm_time"], line='s', ax=ax1)
ax1.set_title('QQ plot for segment_osrm_time')
qqplot(gdf["osrm_time"], line='s', ax=ax2)
ax2.set_title('QQ plot for osrm_time')
plt.suptitle('QQ plots for segment_osrm_time and osrm_time')
plt.show()
```



- The plots above illustrate that the samples deviate from a normal distribution.

Shapiro-Wilk test for normality

H_0 : The sample exhibits normal distribution

H_1 : The sample does not exhibit normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

In [62]:
```
test_stat, p_value = spy.shapiro(gdf_2['osrm_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

In [63]:
```
test_stat, p_value = spy.shapiro(gdf_2['segment_osrm_time'].sample(5000))
print('p-value : ', p_value)
if p_value < 0.05:
    print('The sample does not exhibit normal distribution')
else:
    print('The sample exhibits normal distribution')
```

```
p-value :  0.0
The sample does not exhibit normal distribution
```

Lavene's test for Homogeneity of Variances

In [64]:
```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(gdf_2['osrm_time'], gdf_2['segment_osrm_time'])
print('p-value : ', p_value)

if p_value < 0.05:
    print('The samples do not have Homogenous Variance')
else:
    print('The samples have Homogenous Variance ')
```

```
p-value :  8.349482669010088e-08
The samples do not have Homogenous Variance
```

As the assumptions of T-test are not satisfied, we are performing the non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

In [65]:
```
test_stat, p_value = spy.mannwhitneyu(gdf_2['osrm_time'], gdf_2['segment_osrm_time'])
print('p-value : ', p_value)
if p_value < 0.05:
    print('The samples are not similar')
else:
    print('The samples are similar ')
```

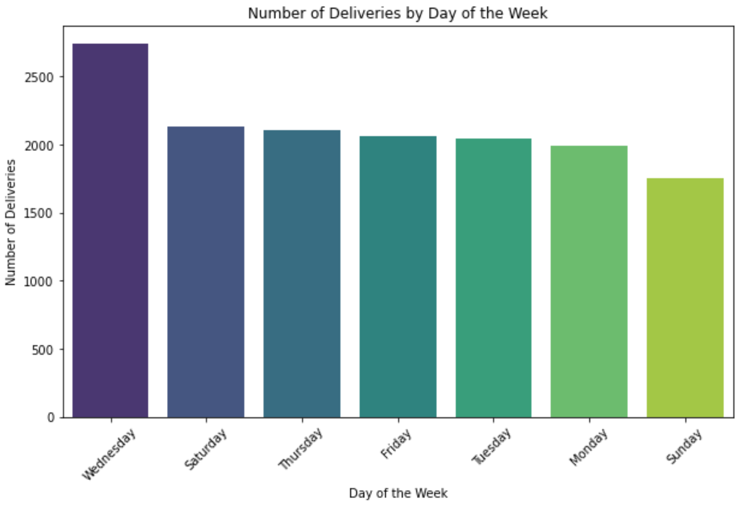
```
p-value :  2.2995370859748865e-08
The samples are not similar
```

Conclusion- osrm_time and segment_osrm_time are not similar since p-value < alpha.

Extra Analysis

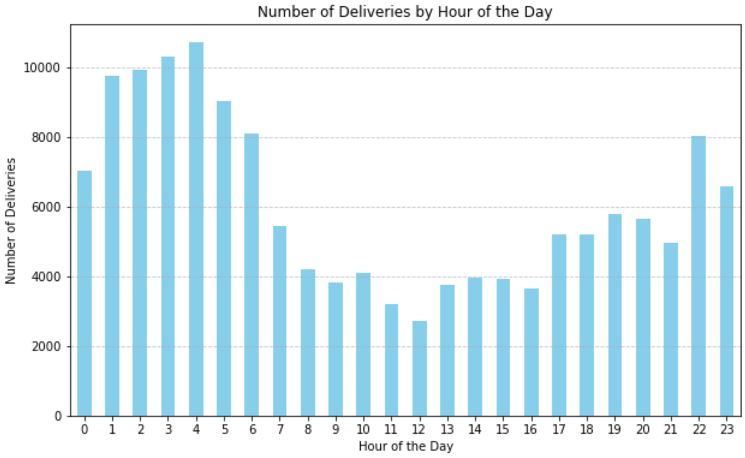
Day of the Week with most deliveries

In [66]:
```
delivery_counts = gdf_2['trip_creation_day_name'].value_counts()

plt.figure(figsize=(10, 6))
sns.barplot(x=delivery_counts.index, y=delivery_counts.values, palette='viridis')
plt.title('Number of Deliveries by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Number of Deliveries')
plt.xticks(rotation=45)
plt.show()
```



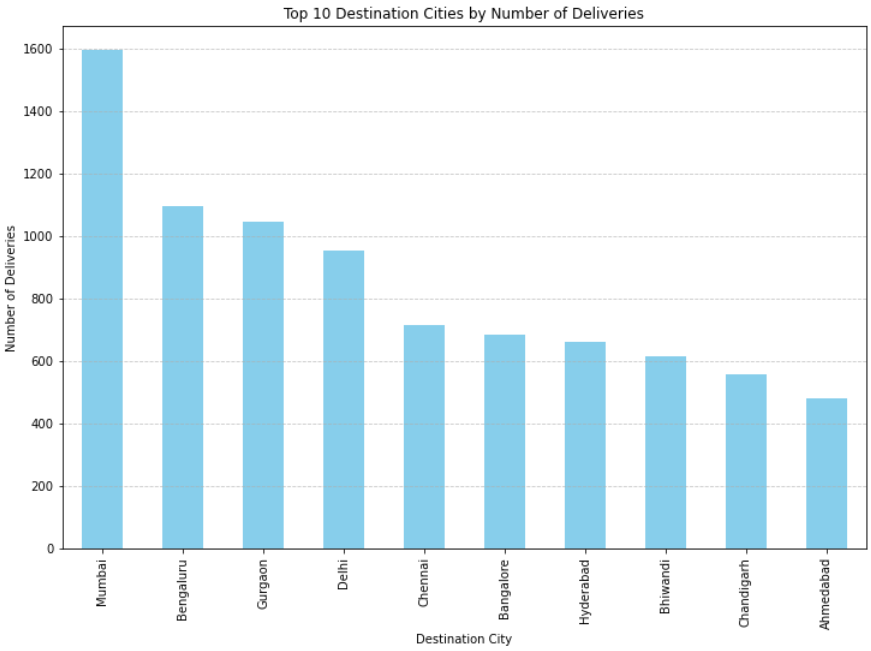
Peak delivery hours

In [67]:
```
df['od_start_time'] = pd.to_datetime(df['od_start_time'])
df['delivery_hour'] = df['od_start_time'].dt.hour
delivery_counts_by_hour = df['delivery_hour'].value_counts().sort_index()

plt.figure(figsize=(10, 6))
delivery_counts_by_hour.plot(kind='bar', color='skyblue')
plt.title('Number of Deliveries by Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Deliveries')
plt.xticks(rotation=0)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

peak_hours = delivery_counts_by_hour[delivery_counts_by_hour == delivery_counts_by_hour.max()].index
print("Peak delivery hour(s):", peak_hours.tolist())
```



Peak delivery hour(s): [4]

In [69]:
```
top_n = 10  # Adjust the number of top cities to display
top_cities = gdf['destination_city'].value_counts().head(top_n)

plt.figure(figsize=(12, 8))
top_cities.plot(kind='bar', color='skyblue')
plt.title(f'Top {top_n} Destination Cities by Number of Deliveries')
plt.xlabel('Destination City')
plt.ylabel('Number of Deliveries')
plt.xticks(rotation=90)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



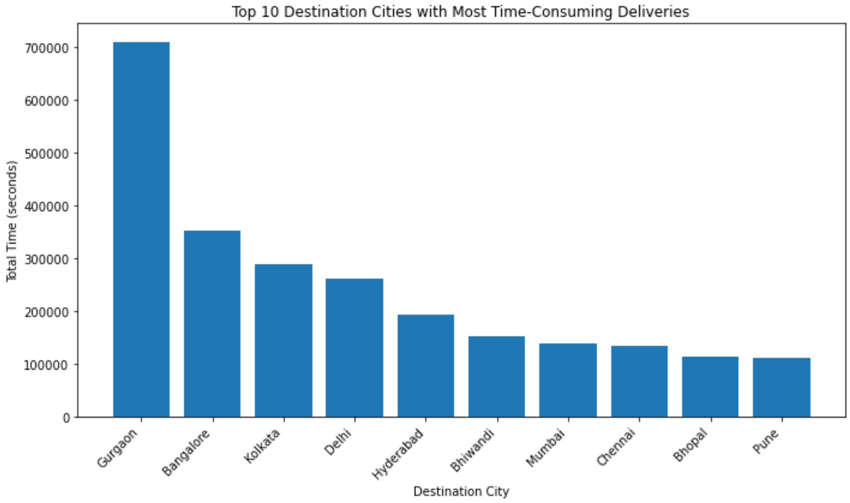
Time period for which the dataset is given

In [70]:
```
df['trip_creation_time'].min(), df['od_end_time'].max()
```

Out[70]:
```
(Timestamp('2018-09-12 00:00:16.535741'),
 Timestamp('2018-10-08 03:00:24.353479'))
```

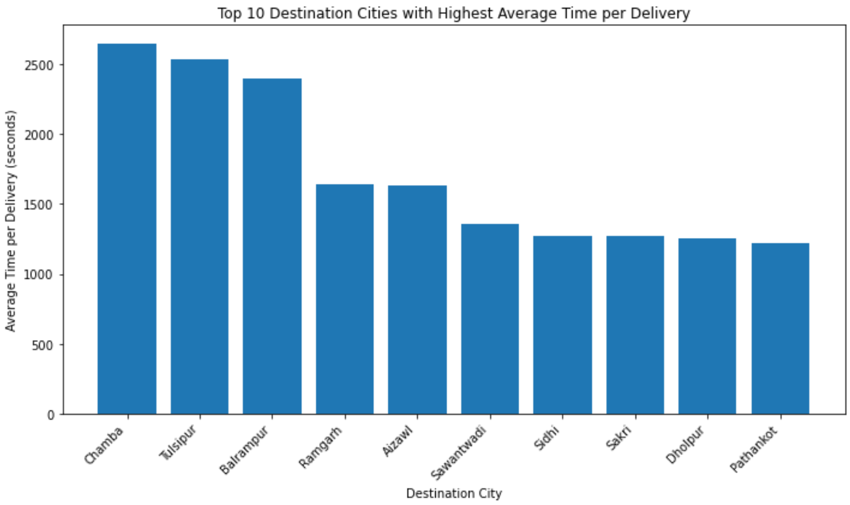
Top 10 Destination Cities with Most Time-Consuming Deliveries

In [71]:
```
time_by_city = gdf_2.groupby('destination_city')['actual_time'].sum().reset_index()
top_10_cities = time_by_city.nlargest(10, 'actual_time')

plt.figure(figsize=(10, 6))
plt.bar(top_10_cities['destination_city'], top_10_cities['actual_time'])
plt.xlabel('Destination City')
plt.ylabel('Total Time (seconds)')
plt.title('Top 10 Destination Cities with Most Time-Consuming Deliveries')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



Top 10 Destination Cities with Highest Average Time per Delivery

In [72]:
```
avg_time_per_delivery = gdf_2.groupby('destination_city')['actual_time'].mean().reset_index()

top_10_cities_avg_time = avg_time_per_delivery.nlargest(10, 'actual_time')

plt.figure(figsize=(10, 6))
plt.bar(top_10_cities_avg_time['destination_city'], top_10_cities_avg_time['actual_time'])
plt.xlabel('Destination City')
plt.ylabel('Average Time per Delivery (seconds)')
plt.title('Top 10 Destination Cities with Highest Average Time per Delivery')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



Most preferable route type

In [73]:
```
sum_time_per_route_type = gdf.groupby('route_type')['actual_time'].sum()
sum_time_per_route_type
```

Out[73]:
```
route_type
Carting    1120662.0
FTL        4171137.0
Name: actual_time, dtype: float64
```

Number of unique values in each column

In [74]:
```
gdf_2.nunique()
```

Out[74]:
```
trip_uuid                          14817
source_center                        938
destination_center                  1042
data                                   2
route_type                             2
trip_creation_time                 14817
source_name                          938
destination_name                    1042
od_total_time                      12655
start_scan_to_end_scan              2208
actual_distance_to_destination     14801
actual_time                         1853
osrm_time                            817
osrm_distance                      14734
segment_actual_time                 1890
segment_osrm_time                   1242
segment_osrm_distance              14754
source_state                          43
source_city                          690
source_place                         761
destination_city                     806
destination_place                    850
destination_state                     39
trip_creation_day                     22
trip_creation_month                    2
trip_creation_year                     1
trip_creation_week                     4
trip_creation_hour                    24
trip_creation_day_name                 7
dtype: int64
```

Identify outliers within the numerical variables and verify their presence through visual analysis.

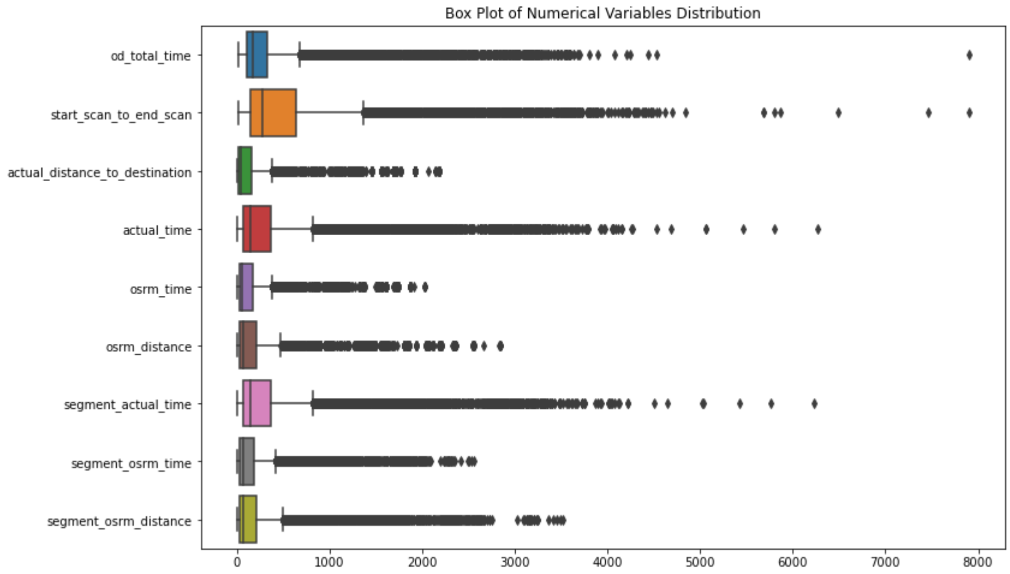
In [75]:
```python
num_columns = ['od_total_time', 'start_scan_to_end_scan', 'actual_distance_to_destination',
                    'actual_time', 'osrm_time', 'osrm_distance', 'segment_actual_time',
                    'segment_osrm_time', 'segment_osrm_distance']
gdf_2[num_columns]
```

Out[75]:

	od_total_time	start_scan_to_end_scan	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	segment_actual_time	segment_osrm_time	segment_osrm_dista
0	1260.60	2259.0	824.732854	1562.0	717.0	991.3523	1548.0	1008.0	1320.4
1	58.83	180.0	73.186911	143.0	68.0	85.1110	141.0	65.0	84.1
2	834.64	3933.0	1927.404273	3347.0	1740.0	2354.0665	3308.0	1941.0	2545.2
3	100.49	100.0	17.175274	59.0	15.0	19.6800	59.0	16.0	19.8
4	152.01	717.0	127.448500	341.0	117.0	146.7918	340.0	115.0	146.7
...	...	...	...	...	...	...	...	...	
14812	152.79	257.0	57.762332	83.0	62.0	73.4630	82.0	62.0	64.8
14813	60.59	60.0	15.513784	21.0	12.0	16.0882	21.0	11.0	16.0
14814	248.41	421.0	38.684839	282.0	48.0	58.9037	281.0	88.0	104.8
14815	105.66	347.0	134.723836	264.0	179.0	171.1103	258.0	221.0	223.5
14816	287.47	353.0	66.081533	275.0	68.0	80.5787	274.0	67.0	80.5

14817 rows × 9 columns

In [76]:
```python
plt.figure(figsize=(12, 8))
sns.boxplot(data=gdf_2[num_columns], orient='h')
plt.title('Box Plot of Numerical Variables Distribution')
plt.show()
```



- The visualizations above clearly indicate the presence of outliers across all numerical columns, highlighting the necessity for outlier treatment.

In [77]:
```python
# Identifying outliers in the numerical variables using the IQR method

def handle_outliers_iqr_stats(data):
```

```
    Q1 = np.percentile(data, 25)
    Q3 = np.percentile(data, 75)
    IQR = Q3 - Q1
    # Bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data < lower_bound) | (data > upper_bound)]

    return {'Q1': Q1, 'Q3': Q3, 'IQR': IQR, 'LB': lower_bound, 'UB': upper_bound, 'Number of outliers': len(outliers)}

for column in num_columns:
    stats = handle_outliers_iqr_stats(gdf_2[column])
    print(f"Column: {column}")
    for key, value in stats.items():
        print(f"{key}: {value}")
    print()
```

```
Column: od_total_time
Q1: 104.16
Q3: 334.75
IQR: 230.59
LB: -241.725
UB: 680.635
Number of outliers: 1584

Column: start_scan_to_end_scan
Q1: 149.0
Q3: 637.0
IQR: 488.0
LB: -583.0
UB: 1369.0
Number of outliers: 1267

Column: actual_distance_to_destination
Q1: 22.83723905859321
Q3: 164.58320763841138
IQR: 141.74596857981817
LB: -189.78171381113404
UB: 377.2021605081386
Number of outliers: 1449

Column: actual_time
Q1: 67.0
Q3: 370.0
IQR: 303.0
LB: -387.5
UB: 824.5
Number of outliers: 1643

Column: osrm_time
Q1: 29.0
Q3: 168.0
IQR: 139.0
LB: -179.5
UB: 376.5
Number of outliers: 1517

Column: osrm_distance
Q1: 30.8192
Q3: 208.475
IQR: 177.6558
LB: -235.6645
UB: 474.9587
Number of outliers: 1524

Column: segment_actual_time
Q1: 66.0
Q3: 367.0
IQR: 301.0
LB: -385.5
UB: 818.5
Number of outliers: 1643

Column: segment_osrm_time
Q1: 31.0
Q3: 185.0
IQR: 154.0
LB: -200.0
UB: 416.0
Number of outliers: 1492

Column: segment_osrm_distance
Q1: 32.6545
Q3: 218.8024
IQR: 186.1479
LB: -246.56735000000003
UB: 498.02425000000005
Number of outliers: 1548
```

Do one-hot encoding of categorical variables (like route_type)

The two most convenient pick for one-hot encoding are categorical variables are route_type' and 'data'.

We start by doing -

- value counts, then
- performing one-hot encoding, finally
- value counts after one-hot encoding.

In [78]:
```
gdf_2['route_type'].value_counts()
```

Out[78]:
```
Carting    8908
FTL        5909
Name: route_type, dtype: int64
```

In [79]:
```
label_encoder = LabelEncoder()
gdf_2['route_type'] = label_encoder.fit_transform(gdf_2['route_type'])
```

In [80]:
```
gdf_2['route_type'].value_counts()
```

Out[80]:
```
0    8908
1    5909
Name: route_type, dtype: int64
```

In [81]:
```
gdf_2['route_type']
```

Out[81]:
```
0        1
1        0
2        1
3        0
4        1
        ..
14812    0
14813    0
14814    0
14815    0
14816    1
Name: route_type, Length: 14817, dtype: int32
```

In [82]:
```
gdf_2['data'].value_counts()
```

Out[82]:
```
training    10654
test         4163
Name: data, dtype: int64
```

In [83]:
```
label_encoder = LabelEncoder()
gdf_2['data'] = label_encoder.fit_transform(gdf_2['data'])
```

In [84]:
```
gdf_2['data'].value_counts()
```

Out[84]:
```
1    10654
0     4163
Name: data, dtype: int64
```

In [85]:
```
gdf_2.head()
```

Out[85]:

	trip_uuid	source_center	destination_center	data	route_type	trip_creation_time	source_name	destination_name	od_total_time	start_scan_to_end_s
0	trip-153671041653548748	IND209304AAA	IND209304AAA	1	1	2018-09-12 00:00:16.535741	Kanpur_Central_H_6 (Uttar Pradesh)	Kanpur_Central_H_6 (Uttar Pradesh)	1260.60	22!
1	trip-153671042288605164	IND561203AAB	IND561203AAB	1	0	2018-09-12 00:00:22.886430	Doddablpur_ChikaDPP_D (Karnataka)	Doddablpur_ChikaDPP_D (Karnataka)	58.83	18
2	trip-153671043369099517	IND000000ACB	IND000000ACB	1	1	2018-09-12 00:00:33.691250	Gurgaon_Bilaspur_HB (Haryana)	Gurgaon_Bilaspur_HB (Haryana)	834.64	39
3	trip-153671046011330457	IND400072AAB	IND401104AAA	1	0	2018-09-12 00:01:00.113710	Mumbai Hub (Maharashtra)	Mumbai_MiraRd_IP (Maharashtra)	100.49	10
4	trip-153671052974046625	IND583101AAA	IND583119AAA	1	1	2018-09-12 00:02:09.740725	Bellary_Dc (Karnataka)	Sandur_WrdN1DPP_D (Karnataka)	152.01	7

5 rows × 29 columns

Normalize/ Standardize the numerical features using MinMaxScaler or StandardScaler.

Normalisation using MinMaxScaler

In [86]:
```
# Create MinMaxScaler object
min_max_scaler = MinMaxScaler()

# Normalize numerical features using MinMaxScaler
# Assuming df is your DataFrame containing the numerical features
gdf_normalized_minmax = gdf_2.copy()  # Make a copy of the original DataFrame
num_columns = ['od_total_time', 'start_scan_to_end_scan', 'actual_distance_to_destination',
                    'actual_time', 'osrm_time', 'osrm_distance', 'segment_actual_time',
                    'segment_osrm_time', 'segment_osrm_distance']

gdf_normalized_minmax[num_columns] = min_max_scaler.fit_transform(gdf_2[num_columns])
---gdf_normalized_minmax[num_columns]
```

Out[86]:

	od_total_time	start_scan_to_end_scan	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	segment_actual_time	segment_osrm_time	segment_osrm_dista
0	-0.157145	-0.283937	-0.374613	-0.248242	-0.350938	-0.346972	-0.247388	-0.391712	-0.373
1	-0.004550	-0.019937	-0.029476	-0.021419	-0.030602	-0.026859	-0.021218	-0.023065	-0.021
2	-0.103058	-0.496508	-0.880999	-0.533568	-0.855874	-0.828325	-0.530301	-0.756450	-0.721
3	-0.009839	-0.009778	-0.003753	-0.007992	-0.004442	-0.003747	-0.008037	-0.003909	-0.003
4	-0.016381	-0.088127	-0.054395	-0.053069	-0.054788	-0.048647	-0.053207	-0.042611	-0.039
...	...	...	...	...	...	...	...	...	
14812	-0.016480	-0.029714	-0.022392	-0.011829	-0.027641	-0.022745	-0.011734	-0.021892	-0.015
14813	-0.004773	-0.004698	-0.002990	-0.001918	-0.002962	-0.002478	-0.001929	-0.001955	-0.001
14814	-0.028621	-0.050540	-0.013631	-0.043638	-0.020731	-0.017602	-0.043723	-0.032056	-0.027
14815	-0.010496	-0.041143	-0.057736	-0.040761	-0.085390	-0.057237	-0.040026	-0.084050	-0.061
14816	-0.033581	-0.041905	-0.026213	-0.042519	-0.030602	-0.025258	-0.042598	-0.023847	-0.020

14817 rows × 9 columns

Standardisation using StandardScaler

In [87]:

```
# Create StandardScaler object
standard_scaler = StandardScaler()

# Standardize numerical features using StandardScaler
gdf_standardized = gdf_2.copy()  # Make a copy of the original DataFrame
gdf_standardized[num_columns] = standard_scaler.fit_transform(gdf_2[num_columns])
gdf_standardized[num_columns]
```

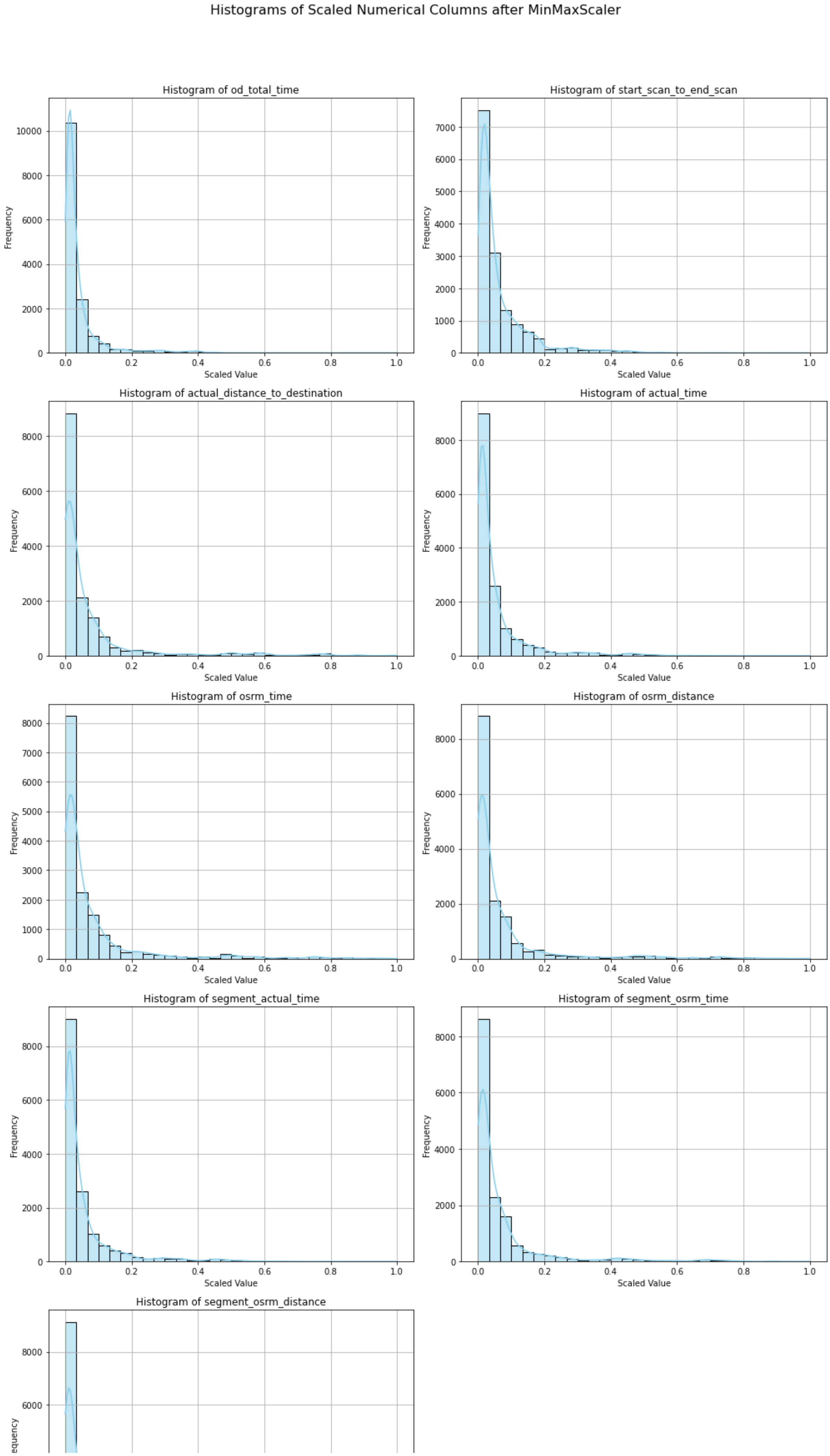
Out[87]:

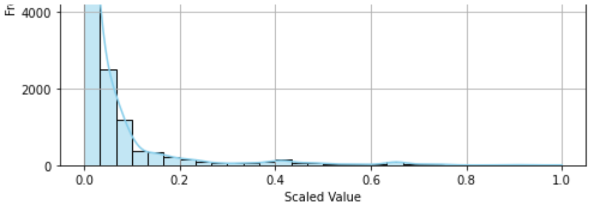
	od_total_time	start_scan_to_end_scan	actual_distance_to_destination	actual_time	osrm_time	osrm_distance	segment_actual_time	segment_osrm_time	segment_osrm_dista
0	1.819659	2.623702	2.162092	2.146251	2.047585	2.124848	2.146791	2.629468	2.633
1	-0.557073	-0.532593	-0.298944	-0.381461	-0.344144	-0.321920	-0.382742	-0.368643	-0.333
2	0.977241	5.165134	5.772935	5.325931	5.817598	5.804050	5.310954	5.595785	5.573
3	-0.474683	-0.654047	-0.482362	-0.531093	-0.539462	-0.498578	-0.530163	-0.524430	-0.488
4	-0.372792	0.282670	-0.121257	-0.028757	-0.163566	-0.155387	-0.024976	-0.209676	-0.183
...	...	...	...	...	...	...	...	...	
14812	-0.371249	-0.415693	-0.349454	-0.488341	-0.366255	-0.353368	-0.488813	-0.378181	-0.380
14813	-0.553593	-0.714774	-0.487802	-0.598784	-0.550518	-0.508275	-0.598480	-0.540327	-0.497
14814	-0.182142	-0.166711	-0.411926	-0.133856	-0.417849	-0.392677	-0.131047	-0.295518	-0.283
14815	-0.464458	-0.279057	-0.097433	-0.165920	0.064919	-0.089730	-0.172397	0.127333	0.000
14816	-0.104894	-0.269947	-0.322212	-0.146325	-0.344144	-0.334157	-0.143632	-0.362284	-0.342

14817 rows × 9 columns

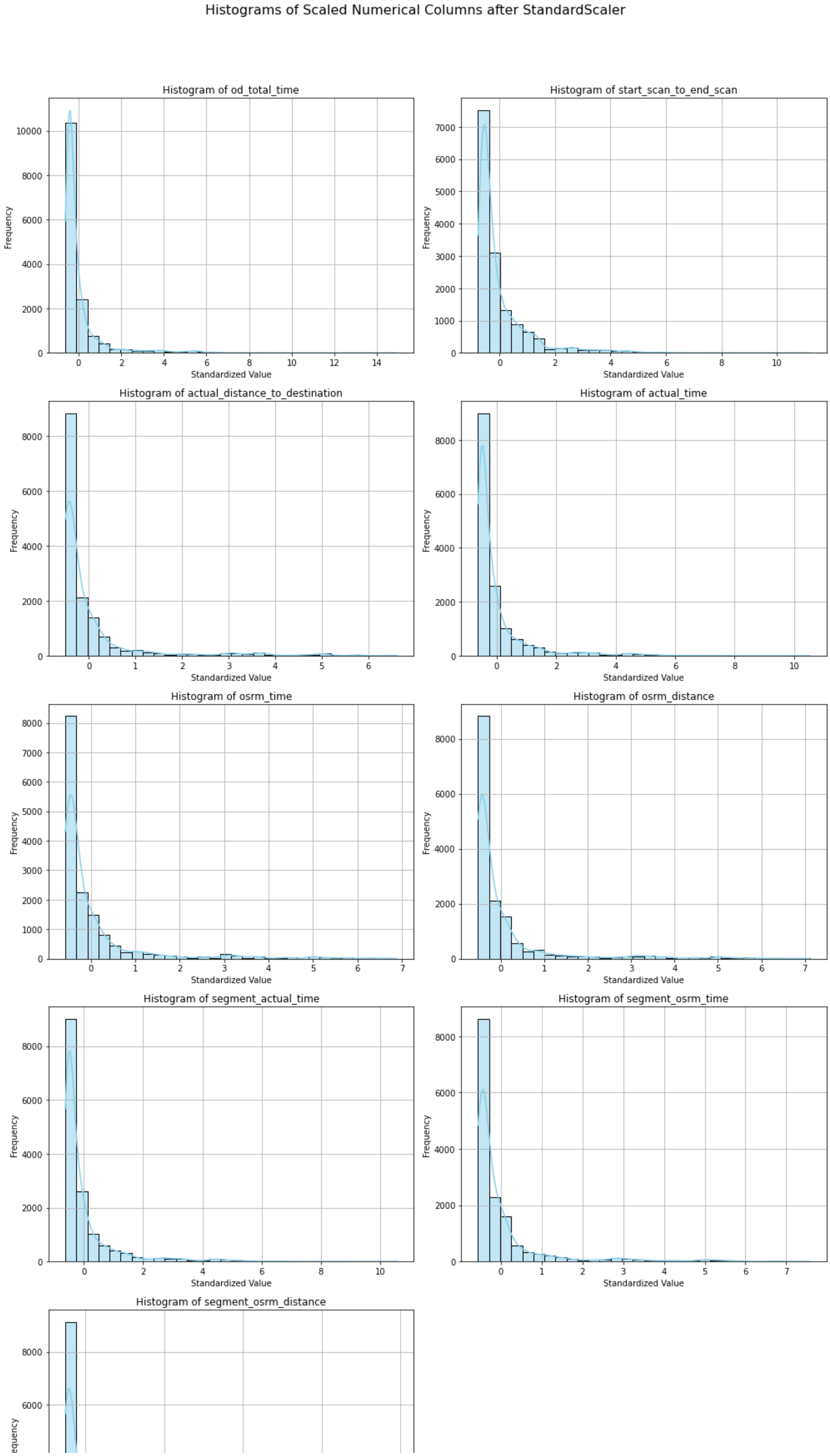
In [88]:

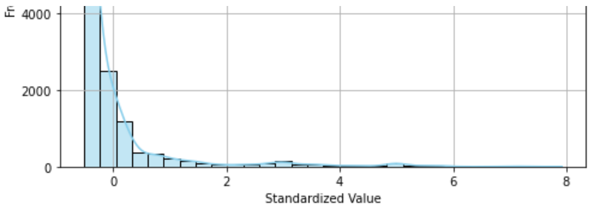
```
plt.figure(figsize=(14, 28))
plt.suptitle('Histograms of Scaled Numerical Columns after MinMaxScaler', fontsize=16)  # Super title

num_plots = len(num_columns)
num_rows = num_plots // 2 + num_plots % 2  # Calculate number of rows needed

for i, column in enumerate(num_columns):
    plt.subplot(num_rows, 2, i + 1)
    sns.histplot(gdf_normalized_minmax[column], bins=30, color='skyblue', kde=True)
    plt.title(f'Histogram of {column}')
    plt.xlabel('Scaled Value')
    plt.ylabel('Frequency')
    plt.grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to make room for super title
plt.show()
```


Histograms of Scaled Numerical Columns after MinMaxScaler
Histogram of od_total_time
Histogram of start_scan_to_end_scan
Histogram of actual_distance_to_destination
Histogram of actual_time
Histogram of osrm_time
Histogram of osrm_distance
Histogram of segment_actual_time
Histogram of segment_osrm_time
Histogram of segment_osrm_distance
Frequency
Scaled Value



In [89]:
```
plt.figure(figsize=(14, 28))
plt.suptitle('Histograms of Scaled Numerical Columns after StandardScaler', fontsize=16)  # Super title

num_plots = len(num_columns)
num_rows = num_plots // 2 + num_plots % 2  # Calculate number of rows needed

for i, column in enumerate(num_columns):
    plt.subplot(num_rows, 2, i + 1)
    sns.histplot(gdf_standardized[column], bins=30, color='skyblue', kde=True)
    plt.title(f'Histogram of {column}')
    plt.xlabel('Standardized Value')
    plt.ylabel('Frequency')
    plt.grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to make room for super title
plt.show()
```


Histograms of Scaled Numerical Columns after StandardScaler
Histogram of od_total_time
Histogram of start_scan_to_end_scan
Histogram of actual_distance_to_destination
Histogram of actual_time
Histogram of osrm_time
Histogram of osrm_distance
Histogram of segment_actual_time
Histogram of segment_osrm_time
Histogram of segment_osrm_distance
Frequency
Standardized Value



Business Insights and Recommendations

Comparison between od_total_time & Start_scan_to_end_scan:

- Hypothesis testing & visual analysis indicate that there is a significant difference between od_total_time & start_scan_to_end_scan. This suggests that the time taken for deliveries varies significantly between these two variables.
- **Recommendation**: We need to further collect data to know factors which are contributing to this difference & to make the delivery process better accordingly.

Comparison between Actual_time & OSRM_time:

- Hypothesis testing & visual analysis indicate that there is a significant difference between Actual_time & OSRM_time. This suggests that the estimated time provided by the OSRM routing engine differs from the actual time taken for deliveries.
- **Recommendation**: Evaluate the accuracy of the OSRM routing engine & consider incorporating additional factors to improve the estimation accuracy.

Comparison between Actual_time & Segment_actual_time:

- While most tests rejected the null hypothesis, the Shapiro-Wilk test for normality showed that samples are similar. This indicates that there might not be a significant difference between Actual_time & Segment_actual_time.
- **Recommendation**: Further investigation is needed to underst& the relationship between these two variables & determine if any optimization opportunities exist.

Comparison between OSRM_distance & Segment_osrm_distance:

- Hypothesis testing & visual analysis indicate that there is a significant difference between OSRM_distance & Segment_osrm_distance. This suggests that the estimated distance provided by the OSRM routing engine differs from the actual distance covered for deliveries.
- **Recommendation**: Look into why there's a difference and think about making the methods used for estimating distances better.

Comparison between OSRM_time & Segment_osrm_time:

- Hypothesis testing & visual analysis indicate that there is a significant difference between OSRM_time & Segment_osrm_time. This suggests that the estimated time provided by the OSRM routing engine differs from the actual time taken for segment deliveries.
- **Recommendation**: Check what things affect how long OSRM thinks a delivery will take. Make improvements to the formulas used by the system based on what you find.

Peak delivery hours occur at 4 AM, while the lowest delivery hours occur at 12 PM.

- **Recommendation**: Make best use of resources & logistics at the time of peak delivery hours as these are the most important hours for the business. If there is less number of employees, then more employees should be put to work.

Day-wise Distribution of Deliveries:

- Most deliveries occur on Wednesdays, while Sundays have the lowest delivery volume. Saturdays have the second-highest delivery volume.
- **Recommendation:** Change the number of workers and materials depending on the delivery trends each day to make operations smoother.

Top Destination Cities with Most Time-Consuming Deliveries:

- Gurugram, Bangalore, Kolkata, & Delhi are the top destination cities with the most time-consuming deliveries.
- **Recommendation**: Examine why deliveries take longer in these cities and put plans in place to make the delivery process smoother.

Most Preferable Route Type:

- Carting route type has significantly more deliveries compared to FTL route type.
- **Recommendation**: Assign resources and plan logistics more effectively according to the preferred route types to improve efficiency and reduce costs.

Outlier Data Analysis:

- Several numerical columns contain outliers, which may affect the overall analysis & interpretation of results.
- **Recommendation**: Take a closer look at the unusual data points to see if they make sense or might be mistakes. Then, decide what to do with them, like changing the data or leaving them out of the analysis.

These insights & recommendations highlight areas for improvement & betterment. By making use of the data-driven insights, businesses can increase their delivery processes, reduce costs, & improve customer satisfaction.