DELHIVERY BIZ CASE STUDY - FEATURE ENGINEERING

About Delhivery

Delhivery is the largest and fastest-growing fully integrated player in India by revenue in Fiscal 2021. They aim to build the operating system for commerce, through a combination of world-class infrastructure, logistics operations of the highest quality, and cutting-edge engineering and technology capabilities.

The Data team builds intelligence and capabilities using this data that helps them to widen the gap between the quality, efficiency, and profitability of their business versus their competitors.

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

Concept Used:

1. Feature Creation
2. Relationship between Features
3. Column Normalization /Column Standardization
4. Handling categorical values
5. Missing values - Outlier treatment / Types of outliers

we need functions and methods to do all these analysis, so we must import Python libraries into our work notebook.

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

To perform Hypothesis testing we need to import few test functions and one hot encoding functions

```
from scipy.stats import ttest_ind,kstest,shapiro
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
```

To get the data into our work space we use the below code(to read csv files) and saving the whole set of data into a single variable(dataframe) which makes analysis easier

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

```
    --2024-01-12 12:51:22--  https://d2beiqkhq929f0.cloudfront.net/public_assets/assets/000/001/551/original/delhivery_data
    Resolving d2beiqkhq929f0.cloudfront.net (d2beiqkhq929f0.cloudfront.net)... 18.155.174.85, 18.155.174.48, 18.155.174.166
    Connecting to d2beiqkhq929f0.cloudfront.net (d2beiqkhq929f0.cloudfront.net)|18.155.174.85|:443... connected.
    HTTP request sent, awaiting response... 200 OK
    Length: 55617130 (53M) [text/plain]
    Saving to: ‘delhivery.csv’

    delhivery.csv       100%[===================>]  53.04M   183MB/s    in 0.3s

    2024-01-12 12:51:22 (183 MB/s) - ‘delhivery.csv’ saved [55617130/55617130]
```

```
df = pd.read_csv('delhivery.csv')
df.head()
```

```
df.sample()
```

TO ANALYSE THE BASIC METRICS/ BASIC STRUCTURE OF THE DATA:

```
# TO GET NO. OF ROWS & COLUMNS:

df.shape
```

```
(144867, 24)
```

```
# TO GET TOTAL ELEMENTS IN THE DATASET (i.e., the dot product of no. of rows & columns)

df.size
```

```
3476808
```

```
# To get index

df.index
```

```
RangeIndex(start=0, stop=144867, step=1)
```

```
# TO GET THE COLUMNS NAMES

df.columns
```

```
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')
```

```
# TO GET THE NAMES OF THE COLUMNS(alternate method)

df.keys()
```

```
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')
```

```
# To get memory usage of each column

df.memory_usage()
```

```
Index                                 128
data                              1158936
trip_creation_time                1158936
route_schedule_uuid               1158936
route_type                        1158936
trip_uuid                         1158936
source_center                     1158936
source_name                       1158936
destination_center                1158936
destination_name                  1158936
od_start_time                     1158936
od_end_time                       1158936
start_scan_to_end_scan            1158936
is_cutoff                          144867
cutoff_factor                     1158936
cutoff_timestamp                  1158936
actual_distance_to_destination    1158936
actual_time                       1158936
osrm_time                         1158936
osrm_distance                     1158936
factor                            1158936
segment_actual_time               1158936
segment_osrm_time                 1158936
segment_osrm_distance             1158936
```

```
segment_factor                    1158936
dtype: int64
```

```
# to get number of unique values in each column

df.nunique()
```

```
data                                   2
trip_creation_time                 14817
route_schedule_uuid                 1504
route_type                             2
trip_uuid                          14817
source_center                       1508
source_name                         1498
destination_center                  1481
destination_name                    1468
od_start_time                      26369
od_end_time                        26369
start_scan_to_end_scan              1915
is_cutoff                              2
cutoff_factor                        501
cutoff_timestamp                   93180
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
```

```
# To get the Time period for which the data is been taken


mini = df['trip_creation_time'].min()
maxi = df['od_end_time'].max()
print(f'start period : {mini}')
print(f'end period : {maxi}')
```

```
     start period : 2018-09-12 00:00:16.535741
     end period : 2018-10-08 03:00:24.353479
```

INFERENCE:

The given data is form the year **2018** and confined from **12th september** and **08th october** months.

```
# TO GET THE STATISTICAL SUMMARY:

df.describe().T
df.describe(include = object).T
```

```
df['data'].value_counts()
```

```
training    104858
test         40009
Name: data, dtype: int64
```

```
df['route_type'].value_counts()
```

```
FTL        99660
Carting    45207
Name: route_type, dtype: int64
```

```
df['source_name'].value_counts()
```

```
Gurgaon_Bilaspur_HB (Haryana)            23347
Bangalore_Nelmngla_H (Karnataka)          9975
Bhiwandi_Mankoli_HB (Maharashtra)         9088
Pune_Tathawde_H (Maharashtra)             4061
Hyderabad_Shamshbd_H (Telangana)          3340
                                         ...  
Shahjhnpur_NavdaCln_D (Uttar Pradesh)        1
Soro_UttarDPP_D (Orissa)                     1
Kayamkulam_Bhrnikvu_D (Kerala)               1
Krishnanagar_AnadiDPP_D (West Bengal)        1
Faridabad_Old (Haryana)                      1
Name: source_name, Length: 1498, dtype: int64
```

```
df['destination_name'].value_counts()
```

```
Gurgaon_Bilaspur_HB (Haryana)        15192
Bangalore_Nelmngla_H (Karnataka)     11019
Bhiwandi_Mankoli_HB (Maharashtra)     5492
Hyderabad_Shamshbd_H (Telangana)      5142
Kolkata_Dankuni_HB (West Bengal)      4892
                                     ...  
Hyd_Trimulgherry_Dc (Telangana)          1
Vijayawada (Andhra Pradesh)              1
Baghpat_Barout_D (Uttar Pradesh)         1
Mumbai_Sanpada_CP (Maharashtra)          1
Basta_Central_DPP_1 (Orissa)             1
Name: destination_name, Length: 1468, dtype: int64
```

INFERENCE FROM RAW DATA:

1. There are more **training**(104858) data
2. **FTL**(Full Truck Load)(99660) - is the most preferred transportation type
3. Most trip is originated at **Gurgaon_Bilaspur_HB (Haryana)** (23347)
4. Majority of the trips are destinated to **Gurgaon_Bilaspur_HB (Haryana)** (15192)

DROPPING ALL THE UNKNOWN FIELDS

The unknown fields has been removed from the original data to perform further analysis.

```
unknown_fields = ['is_cutoff', 'cutoff_factor', 'cutoff_timestamp', 'factor', 'segment_factor']
df.drop(columns = unknown_fields, inplace=True)
```

```
df.columns
```

```
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', 'actual_distance_to_destination',
       'actual_time', 'osrm_time', 'osrm_distance', 'segment_actual_time',
       'segment_osrm_time', 'segment_osrm_distance'],
      dtype='object')
```

MISSING VALUE DETECTION

```
df.isnull().sum()
```

```
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
```

INFERENCE:

There are no null values/missing values in the dataframe except for two columns,

1. source_name - which has 293 missing values
2. destination_name - which has 261 missing values

NUMBER OF UNKNOWN SOURCE & DESTINATION CENTERS:

```
df.loc[df['source_name'].isnull()]['source_center'].nunique()
```

```
10
```

```
df.loc[df['destination_name'].isnull()]['destination_center'].nunique()
```

```
13
```

DEALING WITH NULL VALUES:

```
df['source_name'].fillna('No name',inplace=True)
df['destination_name'].fillna('No name',inplace=True)
```

```
df.isnull().sum()
```

```
data                              0
trip_creation_time                0
route_schedule_uuid               0
route_type                        0
trip_uuid                         0
source_center                     0
source_name                       0
destination_center                0
destination_name                  0
od_start_time                     0
od_end_time                       0
start_scan_to_end_scan            0
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
```

INFERENCE:

The null values have been replaced by 'No name'.

```
df.nunique()
```

```
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
```

```
actual_distance_to_destination    144515
actual_time                         3182
osrm_time                           1531
osrm_distance                     138046
segment_actual_time                  747
segment_osrm_time                    214
segment_osrm_distance             113799
dtype: int64
```

```
# TO GET THE TOTAL INFORMATION ABOUT THE DATASET.
# info function let us know the columns with their data types and no. of non-null values & the total memory usage



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  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                     144867 non-null  object 
 7   destination_center              144867 non-null  object 
 8   destination_name                144867 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: float64(8), object(11)
memory usage: 21.0+ MB
```

INFERENCE:

We can see that,

1. Columns involving time are in different data type
2. Also,columns with 2 unique entries implies that they are categorical

CHANGING THE DTYPE:

```
df[['data','route_type']] = df[['data','route_type']].astype('category')
```

```
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                     144867 non-null  object 
 7   destination_center              144867 non-null  object 
 8   destination_name                144867 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
```

TO CHANGE INTO DATETIME DTYPE:

```
datetime_columns = ['trip_creation_time', 'od_start_time', 'od_end_time']
for element in datetime_columns:
    df[element] = pd.to_datetime(df[element])
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26368 entries, 0 to 26367
Data columns (total 18 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   trip_uuid                       26368 non-null  object
 1   source_center                   26368 non-null  object
 2   destination_center              26368 non-null  object
 3   data                            26368 non-null  category
 4   route_type                      26368 non-null  category
 5   trip_creation_time              26368 non-null  datetime64[ns]
 6   source_name                     26368 non-null  object
 7   destination_name                26368 non-null  object
 8   od_start_time                   26368 non-null  datetime64[ns]
 9   od_end_time                     26368 non-null  datetime64[ns]
 10  start_scan_to_end_scan          26368 non-null  float64
 11  actual_distance_to_destination  26368 non-null  float64
 12  actual_time                     26368 non-null  float64
 13  osrm_time                       26368 non-null  float64
 14  osrm_distance                   26368 non-null  float64
 15  segment_actual_time             26368 non-null  float64
 16  segment_osrm_time               26368 non-null  float64
 17  segment_osrm_distance           26368 non-null  float64
```

```
dtypes: category(2), datetime64[ns](3), float64(8), object(5)
memory usage: 3.3+ MB
```

INFERENCE:

Now, We can clearly visualise the change in dtype of few columns which are more relevant to do analysis

TO MERGE THE ROWS AND AGGREGATING FIELDS

Hint: We can use inbuilt functions like groupby and aggregations like sum(), cumsum() to merge some rows based on their,

1. Trip_uuid, Source ID and Destination ID
2. Further aggregate on the basis of just Trip_uuid. We can also keep the first and last values for some numeric/categorical fields if aggregating them won't make sense.

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

```
df.sample(5)
```

```
df['trip_uuid'].nunique()
```

```
14817
```

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

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

```
0    1260.60
1     999.51
2      58.83
3     122.78
```

```
4     834.64
Name: od_total_time, dtype: float64
```

```
df = df.groupby(by = 'trip_uuid', as_index = False).agg({'source_center' : 'first',
                                                          'destination_center' : 'last',
                                                          'data' : 'first',
                                                          'route_type' : 'first',
                                                          'trip_creation_time' : 'first',
                                                          'source_name' : 'first',
                                                          'destination_name' : 'last',
                                                          'od_total_time' : 'sum',
                                                          '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'})
df.head()
```

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

```
df['source_name'].head(5)
```

```
0    Anand_VUNagar_DC (Gujarat)
1    Anand_VUNagar_DC (Gujarat)
2    Anand_VUNagar_DC (Gujarat)
3    Anand_VUNagar_DC (Gujarat)
4    Anand_VUNagar_DC (Gujarat)
Name: source_name, dtype: object
```

```
def extract_city(x):
    if x == 'No name':
        return 'unknown_city'
    else:
        l = x.split()[0].split('_')
        if 'CCU' in x:
            return 'Kolkata'
        elif 'MAA' in x.upper():
            return 'Chennai'
        elif ('HBR' in x.upper()) or ('BLR' in x.upper()):
            return 'Bengaluru'
        elif 'FBD' in x.upper():
            return 'Faridabad'
        elif 'BOM' in x.upper():
            return 'Mumbai'
        elif 'DEL' in x.upper():
            return 'Delhi'
        elif 'OK' in x.upper():
            return 'Delhi'
        elif 'GZB' in x.upper():
            return 'Ghaziabad'
        elif 'GGN' in x.upper():
            return 'Gurgaon'
        elif 'AMD' in x.upper():
            return 'Ahmedabad'
        elif 'CJB' in x.upper():
            return 'Coimbatore'
        elif 'HYD' in x.upper():
            return 'Hyderabad'
        return l[0]
```

```
def extract_state(x):
  if x == 'No name':
    return 'State not found'
  else:
    temp = x.split('(')
    if len(temp) == 1:
      return temp[0]
    else:
      return temp[1].replace(')','')


def extract_place(x):
    if 'No name' in x:
        return x
    elif 'HBR' in x:
        return 'HBR Layout PC'
    else:
        l = x.split()[0].split('_', 1)
        if len(l) == 1:
            return 'unknown_place'
        else:
            return l[1]
```

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

```
df['source_state'] = df['source_name'].apply(extract_state)

print('Number of source states : ',df['source_state'].nunique())

df['source_state'].unique()
```

```
     Number of source states :  30
     array(['Uttar Pradesh', 'Karnataka', 'Haryana', 'Maharashtra',
            'Tamil Nadu', 'Gujarat', 'Delhi', 'Telangana', 'Rajasthan',
            'Assam', 'Madhya Pradesh', 'West Bengal', 'Andhra Pradesh',
            'Punjab', 'Chandigarh', 'Goa', 'Jharkhand', 'Pondicherry',
            'Orissa', 'Uttarakhand', 'Himachal Pradesh', 'Kerala',
            'Arunachal Pradesh', 'Bihar', 'Chhattisgarh',
            'Dadra and Nagar Haveli', 'Jammu & Kashmir', 'Mizoram', 'Nagaland',
            'State not found'], dtype=object)
```

```
df['source_city'] = df['source_name'].apply(extract_city)
print('No of source cities :', df['source_city'].nunique())
df['source_city'].unique()[:50]
```

```
No of source cities : 690
array(['Kanpur', 'Doddablpur', 'Gurgaon', 'Mumbai', 'Bellary', 'Chennai',
       'Bengaluru', 'Surat', 'Delhi', 'Pune', 'Faridabad', 'Shirala',
       'Hyderabad', 'Thirumalagiri', 'Gulbarga', 'Jaipur', 'Allahabad',
       'Guwahati', 'Narsinghpur', 'Shrirampur', 'Madakasira', 'Sonari',
       'Dindigul', 'Jalandhar', 'Chandigarh', 'Deoli', 'Pandharpur',
       'Kolkata', 'Bhandara', 'Kurnool', 'Bhiwandi', 'Bhatinda',
       'RoopNagar', 'Bantwal', 'Lalru', 'Kadi', 'Shahdol', 'Gangakher',
       'Durgapur', 'Vapi', 'Jamjodhpur', 'Jetpur', 'Mehsana', 'Jabalpur',
       'Junagadh', 'Gundlupet', 'Mysore', 'Goa', 'Bhopal', 'Sonipat'],
      dtype=object)
```

```
df['source_place'] = df['source_name'].apply(extract_place)
print('Number of source places : ',df['source_place'].nunique())
df['source_place'].unique()[:50]
```

```
Number of source places :  757
array(['Central_H_6', 'ChikaDPP_D', 'Bilaspur_HB', 'unknown_place', 'Dc',
       'Poonamallee', 'Chrompet_DPC', 'HBR Layout PC', 'Central_D_12',
       'Lajpat_IP', 'North_D_3', 'Balabhgarh_DPC', 'Central_DPP_3',
       'Shamshbd_H', 'Xroad_D', 'Nehrugnj_I', 'Central_I_7',
       'Central_H_1', 'Nangli_IP', 'North', 'KndliDPP_D', 'Central_D_9',
       'DavkharRd_D', 'Bandel_D', 'RTCStand_D', 'Central_DPP_1',
       'KGAirprt_HB', 'North_D_2', 'Central_D_1', 'DC', 'Mthurard_L',
       'Mullanpr_DC', 'Central_DPP_2', 'RajCmplx_D', 'Beliaghata_DPC',
       'RjnaiDPP_D', 'AbbasNgr_I', 'Mankoli_HB', 'DPC', 'Airport_H',
       'Hub', 'Gateway_HB', 'Tathawde_H', 'ChotiHvl_DC', 'Trmltmpl_D',
       'OnkarDPP_D', 'Mehmdpur_H', 'KaranNGR_D', 'Sohagpur_D',
       'Chrompet_L'], dtype=object)
```

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

```
df['destination_state'] = df['destination_name'].apply(extract_state)
print('Number of destination states : ',df['destination_state'].nunique())
df['destination_state'].unique()
```

```
Number of destination states :  32
array(['Uttar Pradesh', 'Karnataka', 'Haryana', 'Maharashtra',
       'Tamil Nadu', 'Gujarat', 'Delhi', 'Telangana', 'Rajasthan',
       'Madhya Pradesh', 'Assam', 'West Bengal', 'Andhra Pradesh',
       'Punjab', 'Chandigarh', 'Dadra and Nagar Haveli', 'Orissa',
       'Bihar', 'Jharkhand', 'Goa', 'Uttarakhand', 'Himachal Pradesh',
       'Kerala', 'Arunachal Pradesh', 'Mizoram', 'Chhattisgarh',
       'Jammu & Kashmir', 'Nagaland', 'Meghalaya', 'Tripura',
       'State not found', 'Daman & Diu'], dtype=object)
```

```
df['destination_city'] = df['destination_name'].apply(extract_city)
print('Number of destination city : ',df['destination_city'].nunique())
df['destination_city'].unique()[:50]
```

```
Number of destination city :  806
array(['Kanpur', 'Doddablpur', 'Gurgaon', 'Mumbai', 'Sandur', 'Chennai',
       'Bengaluru', 'Surat', 'Delhi', 'PNQ', 'Faridabad', 'Ratnagiri',
       'Bangalore', 'Hyderabad', 'Aland', 'Jaipur', 'Satna', 'Guwahati',
       'Bareli', 'Nashik', 'Hooghly', 'Sivasagar', 'Palani', 'Jalandhar',
       'Chandigarh', 'Yavatmal', 'Sangola', 'Kolkata', 'Savner',
       'Kurnool', 'Bhatinda', 'Bhiwandi', 'Barnala', 'Murbad', 'Kadaba',
       'Gulbarga', 'Naraingarh', 'Ludhiana', 'Kadi', 'Jabalpur',
       'Gangakher', 'Bankura', 'Silvassa', 'Porbandar', 'Jetpur',
       'Khammam', 'Mehsana', 'Katni', 'Una', 'Malavalli'], dtype=object)
```

```
df['destination_place'] = df['destination_name'].apply(extract_place)
print('Number of destination place : ',df['destination_place'].nunique())
df['destination_place'].unique()[:50]
```

```
Number of destination place :  843
array(['Central_H_6', 'ChikaDPP_D', 'Bilaspur_HB', 'MiraRd_IP',
       'WrdN1DPP_D', 'Poonamallee', 'Vandalur_Dc', 'HBR Layout PC',
       'Central_D_3', 'Bhogal', 'unknown_place', 'MjgaonRd_D',
       'Nelmngla_H', 'Uppal_I', 'RazaviRd_D', 'Central_I_7',
       'Central_I_2', 'Hub', 'SourvDPP_D', 'Varachha_DC', 'TgrniaRD_I',
       'DC', 'Gokulam_D', 'Babupaty_D', 'Bomsndra_HB', 'Alwal_I',
       'RjndraRd_D', 'Mehmdpur_H', 'Sanpada_I', 'JajuDPP_D',
       'Central_DPP_2', 'Dankuni_HB', 'Wagodha_D', 'AbbasNgr_I',
       'Balabhgarh_DPC', 'DPC', 'Mankoli_HB', 'Shamshbd_H', 'SnkunDPP_D',
       'Kharar_DC', 'AnugrDPP_D', 'Nehrugnj_I', 'Ward2DPP_D',
       'MilrGanj_HB', 'KaranNGR_D', 'Adhartal_IP', 'Poonamallee_HB',
       'Busstand_D', 'BhowmDPP_D', 'Samrvrni_D'], dtype=object)
```

3. Trip_creation_time: Extract features like month, year and day etc

```
df['trip_creation_time'].head(2)
```

```
0   2018-09-12 00:00:16.535741
1   2018-09-12 00:00:22.886430
Name: trip_creation_time, dtype: datetime64[ns]
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14817 entries, 0 to 14816
Data columns (total 23 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   trip_uuid                       14817 non-null  object
 1   source_center                   14817 non-null  object
 2   destination_center              14817 non-null  object
 3   data                            14817 non-null  category
 4   route_type                      14817 non-null  category
 5   trip_creation_time              14817 non-null  datetime64[ns]
 6   source_name                     14817 non-null  object
 7   destination_name                14817 non-null  object
 8   od_total_time                   14817 non-null  float64
 9   start_scan_to_end_scan          14817 non-null  float64
 10  actual_distance_to_destination  14817 non-null  float64
 11  actual_time                     14817 non-null  float64
 12  osrm_time                       14817 non-null  float64
 13  osrm_distance                   14817 non-null  float64
 14  segment_actual_time             14817 non-null  float64
 15  segment_osrm_time               14817 non-null  float64
 16  segment_osrm_distance           14817 non-null  float64
 17  source_state                    14817 non-null  object
 18  source_city                     14817 non-null  object
 19  source_place                    14817 non-null  object
 20  destination_state               14817 non-null  object
 21  destination_city                14817 non-null  object
 22  destination_place               14817 non-null  object
dtypes: category(2), datetime64[ns](1), float64(9), object(11)
memory usage: 2.4+ MB
```

```
df['trip_creation_year'] = df['trip_creation_time'].dt.year
df['trip_creation_month'] = df['trip_creation_time'].dt.month
df['trip_creation_day'] = df['trip_creation_time'].dt.day
df['trip_creation_week'] = df['trip_creation_time'].dt.isocalendar().week
df['trip_creation_week'] = df['trip_creation_week'].astype('int8')
```

```
df['trip_creation_hour'] = df['trip_creation_time'].dt.hour
df['trip_creation_time'] = df['trip_creation_time'].dt.strftime('%H:%M:%S')
```

Data after cleaning:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14817 entries, 0 to 14816
Data columns (total 28 columns):
 #   Column                          Non-Null Count  Dtype   
---  ------                          --------------  -----   
 0   trip_uuid                       14817 non-null  object  
 1   source_center                   14817 non-null  object  
 2   destination_center              14817 non-null  object  
 3   data                            14817 non-null  category
 4   route_type                      14817 non-null  category
 5   trip_creation_time              14817 non-null  object  
 6   source_name                     14817 non-null  object  
 7   destination_name                14817 non-null  object  
 8   od_total_time                   14817 non-null  float64 
 9   start_scan_to_end_scan          14817 non-null  float64 
 10  actual_distance_to_destination  14817 non-null  float64 
 11  actual_time                     14817 non-null  float64 
 12  osrm_time                       14817 non-null  float64 
 13  osrm_distance                   14817 non-null  float64 
 14  segment_actual_time             14817 non-null  float64 
 15  segment_osrm_time               14817 non-null  float64 
 16  segment_osrm_distance           14817 non-null  float64 
 17  source_state                    14817 non-null  object  
 18  source_city                     14817 non-null  object  
 19  source_place                    14817 non-null  object  
 20  destination_state               14817 non-null  object  
 21  destination_city                14817 non-null  object  
 22  destination_place               14817 non-null  object  
 23  trip_creation_year              14817 non-null  int64   
 24  trip_creation_month             14817 non-null  int64   
 25  trip_creation_day               14817 non-null  int64   
 26  trip_creation_week              14817 non-null  int8    
 27  trip_creation_hour              14817 non-null  int64   
dtypes: category(2), float64(9), int64(4), int8(1), object(12)
memory usage: 2.9+ MB
```

```
df.shape
```

```
(14817, 28)
```

```
df.describe().T
```

```
df.describe(include = object).T
```

TO GET MAX AND MIN PERIOD OF THE TRIP CREATION(ANALYSING MONTHS,WEEK, DAY, HOUR)

ANALYSIS BASED ON DAYS OF THE MONTH

```
df['trip_creation_day'].unique()
```

```
    array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
           29, 30,  1,  2,  3])
```

```
df_day = df.groupby(by = 'trip_creation_day')['trip_uuid'].count().reset_index()
df_day.sample(3)
```

```
plt.figure(figsize = (12, 4))
sns.lineplot(data = df_day,
             x = df_day['trip_creation_day'],
             y = df_day['trip_uuid'])
plt.xticks(np.arange(1, 32))
plt.title('DAY-WISE ANALYSIS')
plt.grid()
```

INFERENCE:

Most trips are created in the **middle** of the month.

ANALYSIS BASED ON WEEK

```
df['trip_creation_week'].unique()
```

```
     array([37, 38, 39, 40], dtype=int8)
```

```
df_week = df.groupby(by = 'trip_creation_week')['trip_uuid'].count().to_frame().reset_index()
df_week.head()
```

```
plt.figure(figsize = (12, 3))
sns.lineplot(data = df_week,
             x = df_week['trip_creation_week'],
             y = df_week['trip_uuid'])
plt.title('WEEKLY ANALYSIS')
plt.grid()
```

INFERENCE:

Maximum trips are created in the **38th** week

ANALYSIS BASED ON THE HOURS OF THE DAY

```
df['trip_creation_hour'].unique()
```

```
    array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
           17, 18, 19, 20, 21, 22, 23])
```

```
df_hour = df.groupby(by = 'trip_creation_hour')['trip_uuid'].count().reset_index()
df_hour.head()
```

```
plt.figure(figsize = (12, 4))
sns.lineplot(data = df_hour,
             x = df_hour['trip_creation_hour'],
             y = df_hour['trip_uuid'])
plt.xticks(np.arange(0,24))
plt.title('HOUR-WISE ANALYSIS')
plt.grid()
```

INFERENCE:

1. During early hours of the day, the count is good
2. But after 5AM till 12PM the count is reducing
3. After 1pm the count starts gradually increasing
4. Reaches **maximum** count on **10PM**

ANALYSIS BASED ON TYPE OF DATA

```
df_data = df.groupby('data')['trip_uuid'].count().reset_index()
df_data['percentage'] = np.round(df_data['trip_uuid'] * 100/ df_data['trip_uuid'].sum(), 2)
df_data.head()
```

```
plt.figure(figsize=(3,3))
plt.pie(x = df_data['trip_uuid'],
        labels = df_data['data'],
        explode = [0, 0.1],
        autopct = '%.2f%%')
plt.title('TEST VS TRAINING')
plt.plot()
```

INFERENCE:

Training data type has maximum trips

ANALYSIS BASED ON ROUTE TYPE

```
df_route = df.groupby(by = 'route_type')['trip_uuid'].count().to_frame().reset_index()
df_route['percentage'] = np.round(df_route['trip_uuid'] * 100/ df_route['trip_uuid'].sum(), 2)
df_route.head()
```

```
plt.figure(figsize=(3,3))
plt.pie(x = df_route['trip_uuid'],
        labels = ['Carting', 'FTL'],
        explode = [0, 0.1],
        autopct = '%.2f%%')
plt.title('ROUTE TYPE ANALYSIS')
plt.plot()
```

INFERENCE:

Carting route type has maximum trips

DISTRIBUTION OF NUMBER OF TRIPS CREATED IN DIFFERENT STATES

```
df.describe(include = object).T
```

```
df_source_state = df.groupby('source_state')['trip_uuid'].count().reset_index()
df_source_state['percentage'] = np.round(df_source_state['trip_uuid'] * 100/ df_source_state['trip_uuid'].sum(), 2)
df_source_state = df_source_state.sort_values(by = 'trip_uuid', ascending = False)
df_source_state.head(2)
```

```
plt.figure(figsize = (10, 15))
sns.barplot(data = df_source_state,
            x = df_source_state['trip_uuid'],
            y = df_source_state['source_state'])
plt.title('Source state - Analysis')
plt.show()
```

INFERENCE:

Maximum trips are originated from **Maharastra** followed by karnataka and Haryana

Distribution of trips created in different cities

```
df_source_city = df.groupby('source_city')['trip_uuid'].count().reset_index()
df_source_city['percentage'] = np.round(df_source_city['trip_uuid'] * 100/ df_source_city['trip_uuid'].sum(), 2)
df_source_city = df_source_city.sort_values(by = 'trip_uuid', ascending = False)[:30]
df_source_city.head(2)
```

```
plt.figure(figsize = (10, 10))
sns.barplot(data = df_source_city,
            x = df_source_city['trip_uuid'],
            y = df_source_city['source_city'])
plt.title('Trips oriented in different cities - analysis')
plt.show()
```

INFERENCE:

Most trips are originated in **Mumbai** followed by Gurgaon and Delhi

Analysis based on the distribution of number of trips based on destination state

```
df_destination_state = df.groupby('destination_state')['trip_uuid'].count().reset_index()
df_destination_state['percentage'] = np.round(df_destination_state['trip_uuid'] * 100/ df_destination_state['trip_uuid'].su
df_destination_state = df_destination_state.sort_values(by = 'trip_uuid', ascending = False)
df_destination_state.head(2)
```

```
plt.figure(figsize = (10, 10))
sns.barplot(data = df_destination_state,
            x = df_destination_state['trip_uuid'],
            y = df_destination_state['destination_state'])
plt.title('Destination state - Analysis')
plt.show()
```

INFERENCE:

Maximum trips are destined to **Maharastra**

Analysis based on the distribution of number of trips based on destination cities

```
df_destination_city = df.groupby('destination_city')['trip_uuid'].count().reset_index()
df_destination_city['percentage'] = np.round(df_destination_city['trip_uuid'] * 100/ df_destination_city['trip_uuid'].sum()
df_destination_city = df_destination_city.sort_values(by = 'trip_uuid', ascending = False)[:30]
df_destination_city.head(3)
```

```
plt.figure(figsize = (10, 10))
sns.barplot(data = df_destination_city,
            x = df_destination_city['trip_uuid'],
            y = df_destination_city['destination_city'])
plt.title('Destination city - Analysis')
plt.show()
```

INFERENCE:

maximum trips are destined to **Mumbai** city followed by Bengaluru and Gurgaon

PAIRPLOT:

```
numerical_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']
sns.pairplot(data = df,
             vars = numerical_columns,
             kind = 'reg',
             hue = 'route_type',
             markers = '*')
plt.show()
```

```
df_corr = df[numerical_columns].corr()
df_corr
```

```
plt.figure(figsize = (15, 10))
sns.heatmap(data = df_corr, annot = True)
plt.show()
```

INFERENCE:

Very High Correlation (> 0.9) exists between columns all the numerical columns specified above

3. In-depth analysis and feature engineering:

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

STEP-1 : Set up Null Hypothesis

1. **Null Hypothesis (H0)** - od_total_time (Total Trip Time) and start_scan_to_end_scan (Expected total trip time) are same.
2. **Alternate Hypothesis (HA)** - od_total_time (Total Trip Time) and start_scan_to_end_scan (Expected total trip time) are different.

STEP-2 : Checking for basic assumpitons for the hypothesis

Distribution check using **QQ Plot**

Homogeneity of Variances using **Lavene's test**

STEP-3: Define Test statistics; Distribution of T under H0.

If the assumptions of T Test are met then we can proceed performing T Test for independent samples else we will perform the non parametric test equivalent to T Test for independent sample i.e., Mann-Whitney U rank test for two independent samples.

STEP-4: Compute the p-value and fix value of alpha.

We set our ***alpha to be 0.05***

STEP-5: Compare p-value and alpha.

Based on p-value, we will reject or fail to reject H0.

p-val < alpha : Reject H0

p-val > alpha : Fail to reject H0

```
df[['od_total_time', 'start_scan_to_end_scan']].describe().T
```

KDE PLOT:

```
plt.figure(figsize=(3,3))
sns.kdeplot(df['od_total_time'])
sns.kdeplot(df['start_scan_to_end_scan'])
plt.show()
```

INFERENCE:

The **kdeplot** vividly shows that the graphs of both the groups are the **same distribution** and they have **almost same mean.** so ttest is performed.

TTEST:

```
# H0 - od_total_time and start_scan_to_end_scan are similar
# Ha - od_total_time and start_scan_to_end_scan are different

test_stat, p_value = spy.ttest_ind(df['od_total_time'],df['start_scan_to_end_scan'])

print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - od_total_time and start_scan_to_end_scan are different')

else:
  print('FAIL TO REJECT H0 - od_total_time and start_scan_to_end_scan are similar')
```

```
P-value : 0.9076773879740099
FAIL TO REJECT H0 - od_total_time and start_scan_to_end_scan are similar
```

Visual Tests to know if the samples follow normal distribution

QQPLOT:

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for od_total_time and start_scan_to_end_scan')
spy.probplot(df['od_total_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for od_total_time')
plt.subplot(1, 2, 2)
spy.probplot(df['start_scan_to_end_scan'], plot = plt, dist = 'norm')
plt.title('QQ plot for start_scan_to_end_scan')
plt.show()
```

It can be seen from the above plots that the samples do not come from normal distribution.

Applying Shapiro-Wilk test for normality

H0 : The sample follows normal distribution

Ha : The sample does not follow normal distribution

alpha = 0.05

Test Statistics : Shapiro-Wilk test for normality

SHAPIRO-WILK TEST:

```
test_stat, p_value = shapiro(df['od_total_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = shapiro(df['start_scan_to_end_scan'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

KS test for normality

```
test_stat, p_value = kstest(df['od_total_time'].sample(5000),spy.norm.cdf)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = kstest(df['start_scan_to_end_scan'].sample(5000),spy.norm.cdf)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

Transforming the data using boxcox transformation to check if the transformed data follows normal distribution.

BOXCOX TRANSFORMATION:

```
transformed_od_total_time = spy.boxcox(df['od_total_time'])[0]
test_stat, p_value = spy.shapiro(transformed_od_total_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 7.172770042757021e-25
REJECT H0 -The sample does not follow normal distribution
/usr/local/lib/python3.10/dist-packages/scipy/stats/_morestats.py:1882: UserWarning: p-value may not be accurate for N
  warnings.warn("p-value may not be accurate for N > 5000.")
```

```
transformed_start_scan_to_end_scan = spy.boxcox(df['start_scan_to_end_scan'])[0]
test_stat, p_value = spy.shapiro(transformed_start_scan_to_end_scan)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 1.0471322892609475e-24
REJECT H0 -The sample does not follow normal distribution
```

For all the test, the sample doesnt follow Gaussian distribution

Homogeneity of Variances using Lavene's test

LAVENE'S TEST:

```
# H0 - Homogenous Variance

# HA - Non Homogenous Variance

test_stat, p_value = spy.levene(df['od_total_time'], df['start_scan_to_end_scan'])
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The samples do not have  Homogenous Variance')
else:
    print('FAIL TO REJECT H0 - The samples have Homogenous Variance ')
```

```
p-value 0.9668007217581142
FAIL TO REJECT H0 - The samples have Homogenous Variance
```

Since the samples are not normally distributed, T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., **Mann-Whitney U rank** test for two independent samples.

Mann-Whitney U rank test:

```
# H0 - od_total_time and start_scan_to_end_scan are similar
# Ha - od_total_time and start_scan_to_end_scan are different


test_stat, p_value = spy.mannwhitneyu(df['od_total_time'], df['start_scan_to_end_scan'])
print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - od_total_time and start_scan_to_end_scan are different')

else:
  print('FAIL TO REJECT H0 - od_total_time and start_scan_to_end_scan are similar')
```

```
P-value : 0.7815123224221716
FAIL TO REJECT H0 - od_total_time and start_scan_to_end_scan are similar
```

INFERENCE:

od_total_time and start_scan_to_end_scan are **similar**

Do hypothesis testing / visual analysis 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)

```
df[['actual_time', 'osrm_time']].describe().T
```

KDEPLOT:

```
plt.figure(figsize=(3,3))
sns.kdeplot(df['actual_time'])
sns.kdeplot(df['osrm_time'])
plt.show()
```

INFERRENCE:

The **kdeplot** vividly shows that the graphs of both the groups are almost the **same distribution** and they have **almost same mean.** so ttest is performed.

TTEST-IND:

```
# H0 - actual_time and osrm_time are similar
# Ha - actual_time and osrm_time are different

test_stat, p_value = spy.ttest_ind(df['actual_time'],df['osrm_time'])

print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - actual_time and osrm_time are different')

else:
  print('FAIL TO REJECT H0 - actual_time and osrm_time are similar')
```

```
P-value : 0.0
REJECT H0 - actual_time and osrm_time are different
```

Visual Tests to know if the samples follow normal distribution

HISTPLOT:

```
plt.figure(figsize = (10, 4))
sns.histplot(df['actual_time'], element = 'step', color = 'green')
sns.histplot(df['osrm_time'], element = 'step', color = 'lightblue')
plt.legend(['actual_time', 'osrm_time'])
plt.show()
```

QQ Plot:

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for actual_time and osrm_time')
spy.probplot(df['actual_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for actual_time')
plt.subplot(1, 2, 2)
spy.probplot(df['osrm_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for osrm_time')
plt.show()
```

Shapiro-Wilk test:

```
test_stat, p_value = shapiro(df['actual_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = shapiro(df['osrm_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

BOXCOX TRANSFORMATION:

```
transformed_actual_time = spy.boxcox(df['actual_time'])[0]
test_stat, p_value = spy.shapiro(transformed_actual_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 1.021792743086169e-28
REJECT H0 -The sample does not follow normal distribution
/usr/local/lib/python3.10/dist-packages/scipy/stats/_morestats.py:1882: UserWarning: p-value may not be accurate for N
  warnings.warn("p-value may not be accurate for N > 5000.")
```

```
transformed_osrm_time = spy.boxcox(df['osrm_time'])[0]
test_stat, p_value = spy.shapiro(transformed_osrm_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 -The sample follows normal distribution')
```

```
p-value 3.543600614978861e-35
REJECT H0 -The sample does not follow normal distribution
```

Lavene's test:

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(df['actual_time'], df['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 1.871297993683208e-220
The samples do not have  Homogenous Variance
```

Since the samples do not follow any of the assumptions T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

Mann-Whitney U rank test:

```
test_stat, p_value = spy.mannwhitneyu(df['actual_time'], df['osrm_time'])
print('p-value', p_value)
alpha = 0.05
if p_value < alpha:
  print('REJECT H0 - actual_time and osrm_time are different')

else:
  print('FAIL TO REJECT H0 - actual_time and osrm_time are similar')
```

```
p-value 0.0
REJECT H0 - actual_time and osrm_time are different
```

INFERENCE:

actual_time and osrm_time are **different**

Do hypothesis testing/ visual analysis between actual_time aggregated value and segment actual time aggregated value (aggregated values are the values you’ll get after merging the rows on the basis of trip_uuid)

```
df[['actual_time', 'segment_actual_time']].describe().T
```

KDE PLOT:

```
plt.figure(figsize=(3,3))
sns.kdeplot(df['actual_time'])
sns.kdeplot(df['segment_actual_time'])
plt.show()
```

INFERRENCE:

The **kdeplot** vividly shows that the graphs of both the groups are the **same distribution** and they have **almost same mean.** so ttest is performed.

TTEST:

```
# H0 - actual_time and segment_actual_time are similar
# Ha - actual_time and segment_actual_time are different

test_stat, p_value = spy.ttest_ind(df['actual_time'],df['segment_actual_time'])

print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - actual_time and segment_actual_time are different')

else:
  print('FAIL TO REJECT H0 - actual_time and segment_actual_time are similar')
```

```
P-value : 0.6165138648224772
FAIL TO REJECT H0 - actual_time and segment_actual_time are similar
```

QQ Plot:

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for actual_time and segment_actual_time')
spy.probplot(df['actual_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for actual_time')
plt.subplot(1, 2, 2)
spy.probplot(df['segment_actual_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for segment_actual_time')
plt.plot()
```

Shapiro-Wilk test:

```
test_stat, p_value = spy.shapiro(df['actual_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df['segment_actual_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

BOXCOX TRANSFORMATION:

```
transformed_actual_time = spy.boxcox(df['actual_time'])[0]
test_stat, p_value = spy.shapiro(transformed_actual_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 1.021792743086169e-28
REJECT H0 -The sample does not follow normal distribution
/usr/local/lib/python3.10/dist-packages/scipy/stats/_morestats.py:1882: UserWarning: p-value may not be accurate for N
  warnings.warn("p-value may not be accurate for N > 5000.")
```

```
transformed_segment_actual_time = spy.boxcox(df['segment_actual_time'])[0]
test_stat, p_value = spy.shapiro(transformed_segment_actual_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 5.696120172016859e-29
REJECT H0 -The sample does not follow normal distribution
```

LAVENE'S test:

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(df['actual_time'], df['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
```

Since the samples do not follow any of the assumptions T-Test cannot be applied here, we can perform its non parametric equivalent test i.e., Mann-Whitney U rank test for two independent samples.

Mann-Whitney U rank test:

```
test_stat, p_value = spy.mannwhitneyu(df['actual_time'], df['segment_actual_time'])
print('p-value', p_value)
alpha = 0.05
if p_value < alpha:
  print('REJECT H0 - actual_time and segment_actual_time are different')

else:
  print('FAIL TO REJECT H0 - actual_time and segment_actual_time are similar')
```

```
p-value 0.4164235159622476
FAIL TO REJECT H0 - actual_time and segment_actual_time are similar
```

INFERENCE:

actual_time and segment_actual_time are **similar**

Do hypothesis testing/ visual analysis between osrm distance aggregated value and segment osrm distance aggregated value (aggregated values are the values you’ll get after merging the rows on the basis of trip_uuid)

```
df[['osrm_distance', 'segment_osrm_distance']].describe().T
```

KDE PLOT:

```
plt.figure(figsize=(3,3))
sns.kdeplot(df['osrm_distance'])
sns.kdeplot(df['segment_osrm_distance'])
plt.show()
```

TTEST:

```
# H0 - osrm_distance and segment_osrm_distance are similar
# Ha - osrm_distance and segment_osrm_distance are different

test_stat, p_value = spy.ttest_ind(df['osrm_distance'],df['segment_osrm_distance'])

print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - osrm_distance and segment_osrm_distance are different')

else:
  print('FAIL TO REJECT H0 - osrm_distance and segment_osrm_distance are similar')
```

```
P-value : 3.8426314733537l8e-05
REJECT H0 - osrm_distance and segment_osrm_distance are different
```

QQ Plot:

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for osrm_distance and segment_osrm_distance')
spy.probplot(df['osrm_distance'], plot = plt, dist = 'norm')
plt.title('QQ plot for osrm_distance')
plt.subplot(1, 2, 2)
spy.probplot(df['segment_osrm_distance'], plot = plt, dist = 'norm')
plt.title('QQ plot for segment_osrm_distance')
plt.plot()
```

Shapiro-Wilk test:

```
test_stat, p_value = spy.shapiro(df['osrm_distance'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df['segment_osrm_distance'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

BOXCOX TRANSFORMATION:

```
transformed_osrm_distance = spy.boxcox(df['osrm_distance'])[0]
test_stat, p_value = spy.shapiro(transformed_osrm_distance)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 7.114532433223529e-41
REJECT H0 -The sample does not follow normal distribution
```

```
transformed_segment_osrm_distance = spy.boxcox(df['segment_osrm_distance'])[0]
test_stat, p_value = spy.shapiro(transformed_segment_osrm_distance)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 3.0623432935550394e-38
REJECT H0 -The sample does not follow normal distribution
```

LAVENE'S TEST:

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(df['osrm_distance'], df['segment_osrm_distance'])
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 0.00020976354422600578
The samples do not have  Homogenous Variance
```

Mann-Whitney U rank test:

```
test_stat, p_value = spy.mannwhitneyu(df['osrm_distance'], df['segment_osrm_distance'])
print('p-value', p_value)
alpha = 0.05
if p_value < alpha:
  print('REJECT H0 - osrm_distance and segment_osrm_distance are different')

else:
  print('FAIL TO REJECT H0 - osrm_distance and segment_osrm_distance are similar')
```

```
     p-value 9.511383588276375e-07
     REJECT H0 - osrm_distance and segment_osrm_distance are different
```

INFERENCE:

osrm_distance and segment_osrm_distance are **different**

Do hypothesis testing/ visual analysis between osrm time aggregated value and segment osrm time aggregated value (aggregated values are the values you'll get after merging the rows on the basis of trip_uuid)

```
df[['osrm_time', 'segment_osrm_time']].describe().T
```

KDE PLOT:

```
plt.figure(figsize=(3,3))
sns.kdeplot(df['osrm_time'])
sns.kdeplot(df['segment_osrm_time'])
plt.show()
```

TTEST:

```
# H0 - osrm_time and segment_osrm_time are similar
# Ha - osrm_time and segment_osrm_time are different

test_stat, p_value = spy.ttest_ind(df['osrm_time'],df['segment_osrm_time'])

print('P-value :',p_value)
alpha = 0.05

if p_value < alpha:
  print('REJECT H0 - osrm_time and segment_osrm_time are different')

else:
  print('FAIL TO REJECT H0 - osrm_time and segment_osrm_time are similar')
```

```
P-value : 9.956426798219171e-09
REJECT H0 - osrm_time and segment_osrm_time are different
```

QQ Plot:

```
plt.figure(figsize = (15, 5))
plt.subplot(1, 2, 1)
plt.suptitle('QQ plots for osrm_time and segment_osrm_time')
spy.probplot(df['osrm_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for osrm_time')
plt.subplot(1, 2, 2)
spy.probplot(df['segment_osrm_time'], plot = plt, dist = 'norm')
plt.title('QQ plot for segment_osrm_time')
plt.plot()
```

Shapiro-Wilk test:

```
test_stat, p_value = spy.shapiro(df['osrm_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

```
test_stat, p_value = spy.shapiro(df['segment_osrm_time'].sample(5000))
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 - The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 0.0
REJECT H0 - The sample does not follow normal distribution
```

BOXCOX TRANSFORMATION:

```
transformed_osrm_time = spy.boxcox(df['osrm_time'])[0]
test_stat, p_value = spy.shapiro(transformed_osrm_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 3.543600614978861e-35
REJECT H0 -The sample does not follow normal distribution
```

```
transformed_segment_osrm_time = spy.boxcox(df['segment_osrm_time'])[0]
test_stat, p_value = spy.shapiro(transformed_segment_osrm_time)
print('p-value', p_value)
if p_value < 0.05:
    print('REJECT H0 -The sample does not follow normal distribution')
else:
    print('FAIL TO REJECT H0 - The sample follows normal distribution')
```

```
p-value 4.893250997154572e-34
REJECT H0 -The sample does not follow normal distribution
```

LAVENE'S TEST:

```
# Null Hypothesis(H0) - Homogenous Variance

# Alternate Hypothesis(HA) - Non Homogenous Variance

test_stat, p_value = spy.levene(df['osrm_time'], df['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
```

Mann-Whitney U rank test:

```
test_stat, p_value = spy.mannwhitneyu(df['osrm_time'], df['segment_osrm_time'])
print('p-value', p_value)
alpha = 0.05
if p_value < alpha:
  print('REJECT H0 - osrm_time and segment_osrm_time are different')

else:
  print('FAIL TO REJECT H0 - osrm_time and segment_osrm_time are similar')
```

```
p-value 2.2995370859748865e-08
REJECT H0 - osrm_time and segment_osrm_time are different
```

INFERENCE:

osrm_time and segment_osrm_time are **different**

Find outliers in the numerical variables (you might find outliers in almost all the variables), and check it using visual analysis

```
numerical_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']

df[numerical_columns].describe().T
```

HISTPLOT:

```
plt.figure(figsize = (18,18))
for i in range(len(numerical_columns)):
    plt.subplot(3, 3, i + 1)
    sns.histplot(df[numerical_columns[i]], bins = 1000, kde = True)
    plt.title(f"Distribution of {numerical_columns[i]} column")
plt.show()
```

INFERENCE:

It can be inferred from the above plots that data in all the numerical columns are **right skewed.**

BOX PLOT:

```
plt.figure(figsize = (18, 15))
for i in range(len(numerical_columns)):
    plt.subplot(3, 3, i + 1)
    sns.boxplot(df[numerical_columns[i]])
    plt.title(f"Distribution of {numerical_columns[i]} column")
plt.show()
```

INFERENCE:

It can be clearly seen in the above plots that there are outliers in all the numerical columns that needs to be treated.

Detecting Outliers:

```
for i in numerical_columns:
    Q1 = np.quantile(df[i], 0.25)
    Q3 = np.quantile(df[i], 0.75)
    IQR = Q3 - Q1
    LB = Q1 - 1.5 * IQR
    UB = Q3 + 1.5 * IQR
    outliers = df.loc[(df[i] < LB) | (df[i] > UB)]
    print('Column :', i)
    print(f'Q1 : {Q1}')
    print(f'Q3 : {Q3}')
    print(f'IQR : {IQR}')
    print(f'LB : {LB}')
    print(f'UB : {UB}')
    print(f'Number of outliers : {outliers.shape[0]}')
    print('----------------------------------')
```

```
Column : od_total_time
Q1 : 149.93
Q3 : 638.2
IQR : 488.27000000000004
LB : -582.4750000000001
UB : 1370.605
Number of outliers : 1266
----------------------------------
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
```

INFERENCE:

The outliers present in our sample data can be the true outliers. It's best to remove outliers only when there is a sound reason for doing so. Some outliers represent natural variations in the population, and they should be left as is in the dataset.

Do one-hot encoding of categorical variables

```
df.nunique()
```

```
    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
    actual_distance_to_destination    144515
    actual_time                         3182
    osrm_time                           1531
    osrm_distance                     138046
    segment_actual_time                  747
    segment_osrm_time                    214
    segment_osrm_distance             113799
    dtype: int64
```

Columns with 2 unique values can be encoded. Here, its only data and route type columns.

Route type

```
# Get value counts before one-hot encoding

df['route_type'].value_counts()
```

```
Carting    8908
FTL        5909
Name: route_type, dtype: int64
```

```
# Perform one-hot encoding on categorical column route type

label_encoder = LabelEncoder()
df['route_type'] = label_encoder.fit_transform(df['route_type'])
```

```
# Get value counts after one-hot encoding

df['route_type'].value_counts()
```

```
0    8908
1    5909
Name: route_type, dtype: int64
```

DATA TYPE:

```
# Get value counts of categorical variable 'data' before one-hot encoding

df['data'].value_counts()
```

```
training    10654
test         4163
Name: data, dtype: int64
```

```
# Perform one-hot encoding on categorical variable 'data'

label_encoder = LabelEncoder()
df['data'] = label_encoder.fit_transform(df['data'])


# Get value counts after one-hot encoding

df['data'].value_counts()
```

```
1    10654
0     4163
Name: data, dtype: int64
```

Normalize/ Standardize the numerical features using MinMaxScaler or StandardScaler.

```
plt.figure(figsize = (6, 3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['od_total_time'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['od_total_time']} column")
plt.legend('od_total_time')
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['start_scan_to_end_scan'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['start_scan_to_end_scan']} column")
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['actual_distance_to_destination'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['actual_distance_to_destination']} column")
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['actual_time'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['actual_time']} column")
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['osrm_time'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['osrm_time']} column")
plt.plot()
```

```
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['osrm_distance'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['osrm_distance']} column")
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['segment_actual_time'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['segment_actual_time']} column")
plt.plot()
plt.figure(figsize = (6,3))
scaler = MinMaxScaler()
scaled = scaler.fit_transform(df['segment_osrm_time'].to_numpy().reshape(-1, 1))
sns.histplot(scaled)
plt.title(f"Normalized {df['segment_osrm_time']} column")
plt.plot()
```