Machine Learning Clustering

OPTIMIZING DELIVERY LOGISTICS USING GEO-SPATIAL CLUSTERING

Michael Olu

[Company Name]

Table of Contents

1. Introduction

1.1 Background and Context

With the growth of e-commerce and rising customer expectations, efficient delivery operations have become critical. Companies strive to cut delivery times and costs while increasing customer satisfaction, but traditional logistics solutions sometimes struggle with variables such as location, timing, and product kind. Machine learning, particularly clustering, provides a solution by grouping comparable deliveries and optimizing routes and resources.

1.2 Research Question

This report explores the question:

How can clustering deliveries by location, time, and goods type optimize logistics efficiency through reduced delivery times and operational costs?

1.4 Literature Review

Studies support the use of clustering for logistics optimization. Rahman et al. (2023) combined K-Means with P-Median techniques to optimize delivery points based on road network distances, improving logistical efficiency. view publication.

Mahesh Prabhu et al. (2020) also applied K-Means and hierarchical clustering to reduce shipment costs by grouping supplier logistics spatially. view publication.
This study builds on these findings with a multi-dimensional clustering approach, incorporating location, time, and goods type to enhance delivery logistics.

2. Datasets

I have chosen the **Amazon Delivery Dataset** from Kaggle for this task, consisting of **43,739 rows** detailing various aspects of delivery logistics. Key columns include:

• **Order_ID**: Unique order identifier
• **Agent_Age**: Age of the delivery agent
• **Agent_Rating**: Customer rating of the agent
• **Store and Drop Coordinates**: Latitude and longitude for pickup and delivery locations
• **Order_Date/Time**: Timestamps for orders and pickups
• **Weather and Traffic**: Conditions affecting delivery
• **Vehicle**: Type of delivery vehicle
• **Area**: Delivery location type (e.g., urban)
• **Delivery_Time**: Duration of delivery
• **Category**: Type of goods delivered

The project aims to optimize time and resource efficiency by pre-emptively planning and assigning delivery tasks based on proximity, timing, and type of deliveries, thereby streamlining driver assignments, balancing workloads, and reducing operational costs.

2.2 Ethical, Social, and Legal Considerations

This dataset provides valuable insights but raises some ethical considerations:

- **Privacy**: No PII is included; however, only aggregate data should be used to protect agent identities.
- **Data Rights**: Users must follow Kaggle's licensing terms.
- **Bias**: Potential geographic or demographic biases should be addressed.
- **Employment**: Optimizations may impact employment practices.

In summary, the dataset is a useful resource, but ethical, social, and legal care is needed for responsible use.

3. EDA and Data Preprocessing

Before conducting some exploratory analysis, we need to install and import all necessary libraries needed for manipulation, visualisation and training of the dataset.

3.1 Installing libraries

```
[1]: pip install folium

Requirement already satisfied: folium in /Applications/anaconda3/lib/python3.12/site-packages (0.18.0)
Requirement already satisfied: branca>=0.6.0 in /Applications/anaconda3/lib/python3.12/site-packages (from folium) (0.8.0)
Requirement already satisfied: jinja2>=2.9 in /Applications/anaconda3/lib/python3.12/site-packages (from folium) (3.1.4)
Requirement already satisfied: numpy in /Applications/anaconda3/lib/python3.12/site-packages (from folium) (1.26.4)
Requirement already satisfied: requests in /Applications/anaconda3/lib/python3.12/site-packages (from folium) (2.32.2)
Requirement already satisfied: xyzservices in /Applications/anaconda3/lib/python3.12/site-packages (from folium) (2022.9.0)
Requirement already satisfied: MarkupSafe>=2.0 in /Applications/anaconda3/lib/python3.12/site-packages (from jinja2>=2.9->folium) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /Applications/anaconda3/lib/python3.12/site-packages (from requests->folium) (2.0.
4)
Requirement already satisfied: idna<4,>=2.5 in /Applications/anaconda3/lib/python3.12/site-packages (from requests->folium) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /Applications/anaconda3/lib/python3.12/site-packages (from requests->folium) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in /Applications/anaconda3/lib/python3.12/site-packages (from requests->folium) (2024.6.2)
Note: you may need to restart the kernel to use updated packages.

[2]: pip install reverse_geocoder

Requirement already satisfied: reverse_geocoder in /Applications/anaconda3/lib/python3.12/site-packages (1.5.1)
Requirement already satisfied: numpy>=1.11.0 in /Applications/anaconda3/lib/python3.12/site-packages (from reverse_geocoder) (1.26.4)
Requirement already satisfied: scipy>=0.17.1 in /Applications/anaconda3/lib/python3.12/site-packages (from reverse_geocoder) (1.13.1)
Note: you may need to restart the kernel to use updated packages.

[3]: pip install yellowbrick

Requirement already satisfied: yellowbrick in /Applications/anaconda3/lib/python3.12/site-packages (1.5)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.2 in /Applications/anaconda3/lib/python3.12/site-packages (from yellowbrick) (3.8.4)
Requirement already satisfied: scipy>=1.0.0 in /Applications/anaconda3/lib/python3.12/site-packages (from yellowbrick) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.0 in /Applications/anaconda3/lib/python3.12/site-packages (from yellowbrick) (1.5.2)
Requirement already satisfied: numpy>=1.16.0 in /Applications/anaconda3/lib/python3.12/site-packages (from yellowbrick) (1.26.4)
Requirement already satisfied: cycler>=0.10.0 in /Applications/anaconda3/lib/python3.12/site-packages (from yellowbrick) (0.11.0)
Requirement already satisfied: contourpy>=1.0.1 in /Applications/anaconda3/lib/python3.12/site-packages (from matplotlib!=3.0.0,>=2.0.2->yell
owbrick) (1.2.0)
Requirement already satisfied: fonttools>=4.22.0 in /Applications/anaconda3/lib/python3.12/site-packages (from matplotlib!=3.0.0,>=2.0.2->yel
lowbrick) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /Applications/anaconda3/lib/python3.12/site-packages (from matplotlib!=3.0.0,>=2.0.2->yel
```

Figure 1

In Figure 1, we have installed some libraries which will be useful in subsequent processing of the dataset:

- Folium: For map plotting and visualisations
- Reverse_geocoder: For getting location data based on longitude and latitude
- Yellowbrick: For plotting more descriptive elbow graphs.

3.2 Importing libraries

```
[6]: import reverse_geocoder
from yellowbrick.cluster import KElbowVisualizer
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
import folium
import pandas as pd
import numpy as np
```

Figure 2

These libraries will enable us to perform all required procedures on our dataset towards our final goal.

3.3 Importing and viewing the dataset

```
[7]: dataset = pd.read_csv("amazon_delivery.csv")
```

```
[8]: dataset.head()
```

[8]:

	Order_ID	Agent_Age	Agent_Rating	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude	Order_Date	Order_Time	Pickup_Time	Weathe
0	ialx566343618	37	4.9	22.745049	75.892471	22.765049	75.912471	2022-03-19	11:30:00	11:45:00	Sunn
1	akqg208421122	34	4.5	12.913041	77.683237	13.043041	77.813237	2022-03-25	19:45:00	19:50:00	Storm
2	njpu434582536	23	4.4	12.914264	77.678400	12.924264	77.688400	2022-03-19	08:30:00	08:45:00	Sandstorm
3	rjto796129700	38	4.7	11.003669	76.976494	11.053669	77.026494	2022-04-05	18:00:00	18:10:00	Sunn
4	zguw716275638	32	4.6	12.972793	80.249982	13.012793	80.289982	2022-03-26	13:30:00	13:45:00	Cloud

```
[9]: dataset.tail()
```

[9]:

de	Store_Longitude	Drop_Latitude	Drop_Longitude	Order_Date	Order_Time	Pickup_Time	Weather	Traffic	Vehicle	Area	Delivery_Time	Category
28	75.794257	26.912328	75.804257	2022-03-24	11:35:00	11:45:00	Windy	High	motorcycle	Metropolitian	160	Home
00	0.000000	0.070000	0.070000	2022-02-16	19:55:00	20:10:00	Windy	Jam	motorcycle	Metropolitian	180	Jewelry
94	80.242439	13.052394	80.272439	2022-03-11	23:50:00	00:05:00	Cloudy	Low	scooter	Metropolitian	80	Home
53	76.986241	11.041753	77.026241	2022-03-07	13:35:00	13:40:00	Cloudy	High	motorcycle	Metropolitian	130	Kitchen
58	85.325731	23.431058	85.405731	2022-03-02	17:10:00	17:15:00	Fog	Medium	scooter	Metropolitian	180	Cosmetics

Figure 3

```
[689]: dataset.describe()
```

[689]:

	Agent_Age	Agent_Rating	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude	Delivery_Time
count	43739.000000	43685.000000	43739.000000	43739.000000	43739.000000	43739.000000	43739.000000
mean	29.567137	4.633780	17.210960	70.661177	17.459031	70.821842	124.905645
std	5.815155	0.334716	7.764225	21.475005	7.342950	21.153148	51.915451
min	15.000000	1.000000	-30.902872	-88.366217	0.010000	0.010000	10.000000
25%	25.000000	4.500000	12.933298	73.170283	12.985996	73.280000	90.000000
50%	30.000000	4.700000	18.551440	75.898497	18.633626	76.002574	125.000000
75%	35.000000	4.900000	22.732225	78.045359	22.785049	78.104095	160.000000
max	50.000000	6.000000	30.914057	88.433452	31.054057	88.563452	270.000000

Figure 4

```
[690]: dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 43739 entries, 0 to 43738
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Order_ID         43739 non-null  object
 1   Agent_Age        43739 non-null  int64
 2   Agent_Rating     43685 non-null  float64
 3   Store_Latitude   43739 non-null  float64
 4   Store_Longitude  43739 non-null  float64
 5   Drop_Latitude    43739 non-null  float64
 6   Drop_Longitude   43739 non-null  float64
 7   Order_Date       43739 non-null  object
 8   Order_Time       43739 non-null  object
 9   Pickup_Time      43739 non-null  object
 10  Weather          43648 non-null  object
 11  Traffic          43739 non-null  object
 12  Vehicle          43739 non-null  object
 13  Area             43739 non-null  object
 14  Delivery_Time    43739 non-null  int64
 15  Category         43739 non-null  object
dtypes: float64(5), int64(2), object(9)
memory usage: 5.3+ MB
```

Figure 5

This gives us a general idea of what features are present in the dataset. It shows that there are 43,739 instances in the dataset, made up of objects, integers and float data types.

3.4 Checking for null values

```
[11]: dataset.isnull().sum()
```

```
[11]: Order_ID            0
      Agent_Age           0
      Agent_Rating       54
      Store_Latitude      0
      Store_Longitude     0
      Drop_Latitude       0
      Drop_Longitude      0
      Order_Date          0
      Order_Time          0
      Pickup_Time         0
      Weather            91
      Traffic             0
      Vehicle             0
      Area                0
      Delivery_Time       0
      Category            0
      dtype: int64
```

Figure 6

The result shows that there some are null values contained in the Agent_Rating and Weather columns. Although these features would not be required for the training of our final model, I will perform a data pre-processing steps to fill in these values based on the most frequent values occurring in those columns.

```
[12]: catImputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
      catImputer = catImputer.fit(dataset[['Agent_Rating', 'Weather']])
      dataset[['Agent_Rating', 'Weather']] = catImputer.transform(dataset[['Agent_Rating', 'Weather']])
```

```
[13]: dataset.isnull().sum()
```

```
[13]: Order_ID           0
      Agent_Age          0
      Agent_Rating       0
      Store_Latitude     0
      Store_Longitude    0
      Drop_Latitude      0
      Drop_Longitude     0
      Order_Date         0
      Order_Time         0
      Pickup_Time        0
      Weather            0
      Traffic            0
      Vehicle            0
      Area               0
      Delivery_Time      0
      Category           0
      dtype: int64
```

Figure 7

The output shows that the values have been filled up with the most frequent values after applying the SimpleImputer algorithm to those columns.

3.5 Feature Engineering

Some extra features were created to aid the exploratory analysis to extract some valuable insights from the dataset

```
[427]: # Convert the Order_Time column to datetime.time, handling errors if any occur
       dataset['Pickup_Time'] = pd.to_datetime(dataset['Pickup_Time'], format='%H:%M:%S', errors='coerce').dt.time

[428]: # Defining funtion to extract time of day
       def categorize_time(t):
           if t < pd.to_datetime('06:00:00').time():
               return 'Pre-dawn (00:00 - 06:00)'
           elif t < pd.to_datetime('12:00:00').time():
               return 'Morning (06:00 - 12:00)'
           elif t < pd.to_datetime('18:00:00').time():
               return 'Afternoon (12:00 - 18:00)'
           elif t < pd.to_datetime('23:59:59').time():
               return 'Night (18:00 - 00:00)'

[429]: # Apply the categorize_time function to the dataframe
       dataset['Time_Category'] = dataset['Pickup_Time'].apply(categorize_time)

[430]: # Extract day of the week to new column
       dataset['Order_Date'] = pd.to_datetime(dataset['Order_Date'], format='%Y-%m-%d')
       dataset['Day_of_Week'] = dataset['Order_Date'].dt.day_name()

[431]: dataset.head()
```

[431]:

e	Drop_Longitude	Order_Date	Order_Time	Pickup_Time	Weather	Traffic	Vehicle	Area	Delivery_Time	Category	Time_Category	Day_of_Week
9	75.912471	2022-03-19	11:30:00	11:45:00	Sunny	High	motorcycle	Urban	120	Clothing	Morning (06:00 - 12:00)	Saturday
1	77.813237	2022-03-25	19:45:00	19:50:00	Stormy	Jam	scooter	Metropolitian	165	Electronics	Night (18:00 - 00:00)	Friday
4	77.688400	2022-03-19	08:30:00	08:45:00	Sandstorms	Low	motorcycle	Urban	130	Sports	Morning (06:00 - 12:00)	Saturday
9	77.026494	2022-04-05	18:00:00	18:10:00	Sunny	Medium	motorcycle	Metropolitian	105	Cosmetics	Night (18:00 - 00:00)	Tuesday

Figure 8

The code in figure 6 shows where categories of time of the day were predefined in the categorize_time function and matched to each instance in the dataset.

Days of the week was also engineered to aid in visualisation of dates of each instance. These extra features will come in handy when trying to find out what day of the day of the week or what time of the day each order fall into.

The result shows new columns: Time_Category and Day_of_Week has been created.

```
[499]: # Convertin coordinates columns into 2D arrays and running them against the reverse geocoder
       # reverse_geocoder extracts the country and city names based on the longitude and latitude.
       drop_coordinates_list = list(zip(dataset['Drop_Latitude'], dataset['Drop_Longitude']))
       drop_locations = reverse_geocoder.search(drop_coordinates_list ,mode=1)

       store_coordinates_list = list(zip(dataset['Store_Latitude'], dataset['Store_Longitude']))
       store_locations = reverse_geocoder.search(store_coordinates_list ,mode=1)

[500]: # converting final output from reverse_geocoder into dataframes
       drop_locations_df = pd.DataFrame(drop_locations)
       store_locations_df = pd.DataFrame(store_locations)

[501]: # Renaming column names for more readability
       drop_locations_df.rename(columns={'name': 'Destination_City', 'cc': 'Destination_Country'}, inplace=True)
       store_locations_df.rename(columns={'name': 'Store_City', 'cc': 'Store_Country'}, inplace=True)

[502]: drop_locations_df.head()
```

[502]:

	lat	lon	Destination_City	admin1	admin2	Destination_Country
0	22.71792	75.8333	Indore	Madhya Pradesh	Indore	IN
1	13.0707	77.79814	Hoskote	Karnataka	Bangalore Rural	IN
2	12.97194	77.59369	Bangalore	Karnataka	Bangalore Urban	IN
3	11.01867	77.06624	Irugur	Tamil Nadu	Coimbatore	IN
4	13.00639	80.25417	Gandhi Nagar	Tamil Nadu	Chennai	IN

Figure 9

The code in figure 7 shows how country and city names were extracted based on longitude and latitude coordinates and converted into a separate dataframe.

```
[621]: # joining the location dataframes to the main dataset
       dataset = pd.concat([dataset, store_locations_df_city_country, drop_locations_df_city_country], axis=1)

[622]: # Dropping the unnecessary columns to align with our main aim
       dataset = dataset.drop(columns=["Agent_Age", "Agent_Rating", "Delivery_Time"])

[623]: dataset.head()
```

[623]:

_Time	Weather	Traffic	Vehicle	Area	Category	Time_Category	Day_of_Week	Store_City	Store_Country	Destination_City	Destination_Country
:45:00	Sunny	High	motorcycle	Urban	Clothing	Morning (06:00 - 12:00)	Saturday	Indore	IN	Indore	IN
:50:00	Stormy	Jam	scooter	Metropolitian	Electronics	Night (18:00 - 00:00)	Friday	Bangalore	IN	Hoskote	IN
:45:00	Sandstorms	Low	motorcycle	Urban	Sports	Morning (06:00 - 12:00)	Saturday	Bangalore	IN	Bangalore	IN
:10:00	Sunny	Medium	motorcycle	Metropolitian	Cosmetics	Night (18:00 - 00:00)	Tuesday	Coimbatore	IN	Irugur	IN
:45:00	Cloudy	High	scooter	Metropolitian	Toys	Afternoon (12:00 - 18:00)	Saturday	Perungudi	IN	Gandhi Nagar	IN

Figure 10

The code in figure 8 shows the location dataframes joined to the main dataframe, Agent_age, Agent_Rating and Delivery time columns have been dropped to align with the general aim of the task which is to plan unassigned Deliveries and assign each cluster (Area or time proximity) to different drivers.

```
[679]: def haversine(lat1, lon1, lat2, lon2):
           # Convert decimal degrees to radians
           lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])

           # Haversine formula
           dlat = lat2 - lat1
           dlon = lon2 - lon1
           a = np.sin(dlat / 2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon / 2)**2
           c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))

           # Radius of Earth in kilometers (mean radius)
           r = 6371
           return r * c

[680]: dataset['Distance_km'] = dataset.apply(lambda row: haversine(row['Store_Latitude'], row['Store_Longitude'], row['Drop_Latitude'],
                                                                   row['Drop_Longitude']), axis=1)
```

Figure 11

Distance between the store and delivery locations has also been calculated and stored in a new column using the haversine formula to calculate distance on the spherical earth.

3.6 Visualisations

```
[769]: # Defining duncrion for plotting of a barplot
       def plot_frequency(dataframe, column, range):
           sns.set(style="whitegrid")
           # plt.rcParams['figure.dpi'] = 600
           # plt.rcParams['savefig.dpi'] = 600

           # Count the frequency each distinct value in the column
           if range == 'all':
               category_counts = dataframe[column].value_counts().reset_index()
           else:
               category_counts = dataframe[column].value_counts().head(range).reset_index()
           category_counts.columns = [column, 'Frequency']

           # Create the bar plot
           plt.figure(figsize=(18,6))
           bar_plot = sns.barplot(x=column, y='Frequency', hue=column, data=category_counts, palette='coolwarm', dodge=False, legend=False)

           plt.title(f'Frequency of {column}', fontsize=14)
           plt.xlabel(column, fontsize=12)
           plt.ylabel('Frequency', fontsize=12)

           for p in bar_plot.patches:
               bar_plot.annotate(f'{int(p.get_height())}',
                                 (p.get_x() + p.get_width() / 2., p.get_height()),
                                 ha='center', va='bottom',
                                 fontsize=12, color='black',
                                 xytext=(0, 5),  # Offset text slightly above the bar
                                 textcoords='offset points')
```

Figure 12

The code in figure 12 shows a function which has been defined to facilitate visualisations of a bar plot. It takes in a dataframe, the column to visualise and a range value, e.g.: range of top 10 frequencies.

```
[807]: plot_frequency(dataset, 'Destination_Country', 'all')
```

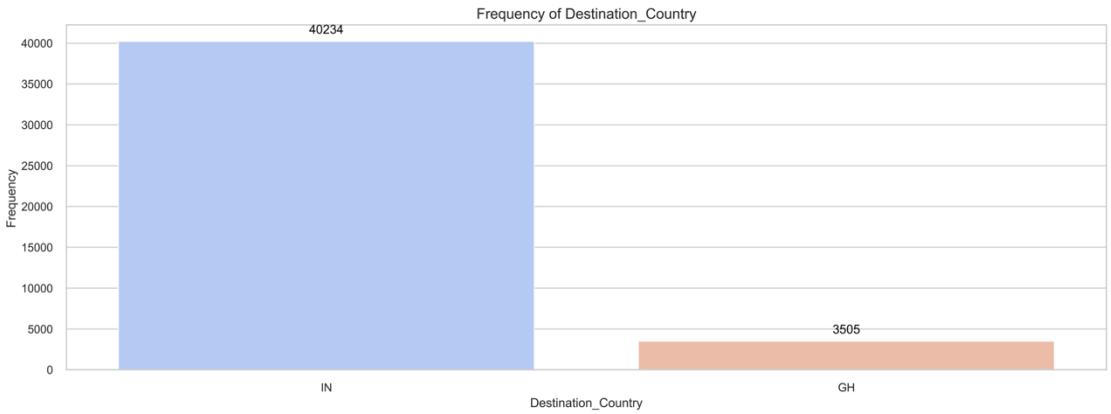


Figure 13

Using the predefined function for bar plot visualisations, the frequency of destination countries has been plotted and there seems to be a distribution among two countries: India and Ghana.

For more context, I decided to also visualise these map coordinates on a map using the function in figure 13 below.

```
[808]: # Defining func
def plot_map(df, cluster_column=None, color=False):
    map_center = [df['Drop_Latitude'].mean(), df['Drop_Longitude'].mean()]
    map_ = folium.Map(location=map_center, zoom_start=11, tiles="OpenStreetMap")

    folium.raster_layers.TileLayer(
        tiles="http://mt1.google.com/vt/lyrs=m&h1=p1Z&x={x}&y={y}&z={z}",
        name="Standard Roadmap",
        attr="Google Map"
    ).add_to(map_)

    if cluster_column == None:
        for index, row in df.iterrows():
            folium.Marker(location=[row['Drop_Latitude'], row['Drop_Longitude']],  # Latitude and Longitude
                          popup=row[[ 'Order_ID', 'Pickup_Time', 'Area', 'Category', 'Time_Category']],  # Popup showing the location name
                          icon=folium.DivIcon(html = '''<i class="fa-solid fa-location-dot" style="color: #f50505; font-size: 15px"></i>''
                         ).add_to(map_)
    else:
        if color:
            for index, row in df.iterrows():
                folium.Marker(location=[row['Drop_Latitude'], row['Drop_Longitude']],  # Latitude and Longitude
                              popup=row[[ 'Order_ID', 'Pickup_Time', 'Area', 'Category', 'Time_Category', "pickup_location_Cluster", "time_Clu
                              icon=folium.DivIcon(html = f'''<i class="fa-solid fa-location-dot" style="color: {row[color]}; font-size: 15px">
                             ).add_to(map_)
        else:
            for index, row in df.iterrows():
                folium.Marker(location=[row['Drop_Latitude'], row['Drop_Longitude']],  # Latitude and Longitude
                              popup=row[[ 'Order_ID', 'Pickup_Time', 'Area', 'Category', 'time_Category', "pickup_location_Cluster"]],  # Popu
                              icon=folium.DivIcon(html = f'''<i class="fa-solid fa-location-dot" style="color: #f50505; font-size: 15px"></i>'
                             ).add_to(map_)


    return map_
```

Figure 14

The plot_map function utilizes the folium library to visualize maps, taking dataframe, cluster_column, and color arguments. It will be used for cluster visualization on the map and will be used for all future map visualizations.

```
[809]: dfr = dataset[dataset["Destination_Country"] == "GH"]
       plot_map(df=dfr)
```

Figure 15

The delivery coordinates for Ghana's locations appear to be inaccurately positioned over the Gulf of Guinea, indicating potential data corruption, thus excluding them from further analysis.

```
[810]: # Removing inaccurate data

       dataset = dataset[dataset["Destination_Country"] != "GH"]
```

```
[811]: plot_frequency(dataset,'Destination_Country', 10)
```

Figure 16

The said instances in Ghana have been dropped and we are left with only deliveries in India.

The next image shows the visualisations for deliveries made in India.

```
[128]: dfr = dataset[dataset["Destination_Country"] == "IN"].sample(100)
       plot_map(df=dfr)
```

[128]:

Figure 17

The coordinates plotted for India appear to be accurate as the map pins are plotted directly on the Indian country map.

```
[129]: plot_frequency(dataset,'Destination_City', 10)
```

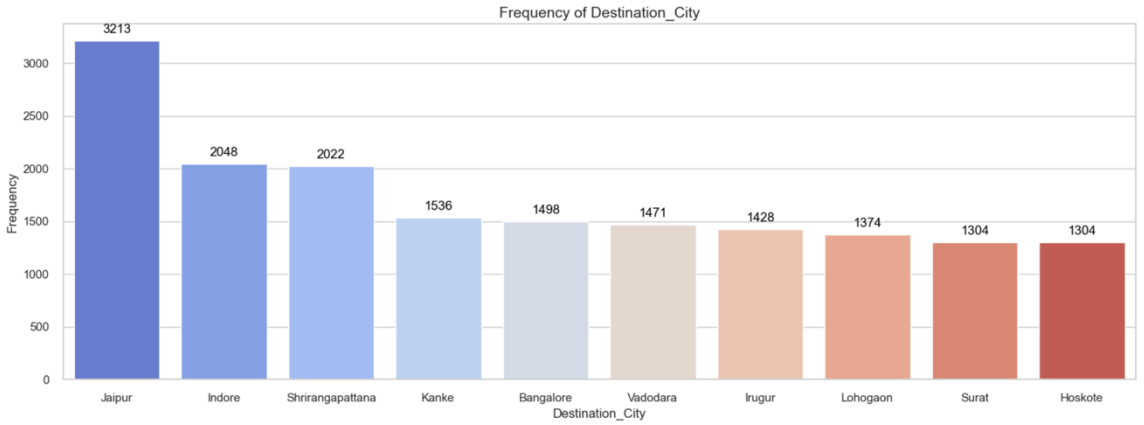


Figure 18

Visualising the frequency bar plot for the top 10 Destination Cities reveals Jaipur with 3,213 instances as the most frequent and Hoskote with 1,304 instances as the 10th most frequent.

```
[131]: plot_frequency(dataset,'Category', 'all')
```

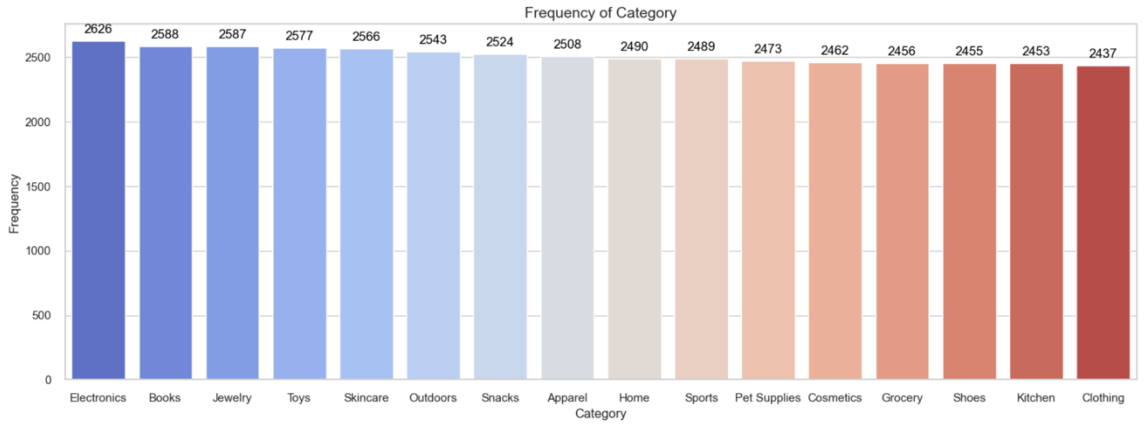


Figure 19

Visualising the Category column reveals that it is almost evenly distributed but with Electronics at 2,626 instances as the most frequent and Clothing with 2,437 instances as the least frequent.

```
[132]: plot_frequency(dataset,'Time_Category', 'all')
```

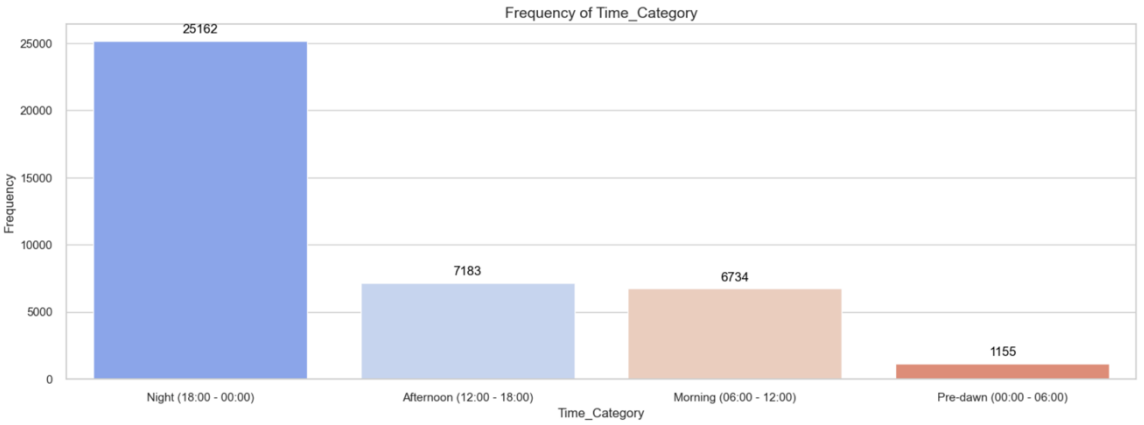


Figure 20

Visualising the time category column which was engineered shows that most delivery pick up times fall during nighttime from 18:00 – 00:00 and the least at predawn hours.

We will also visualise these pick-up time data using a line chart show a more accurate representation of the pick-up hours distributed all through the day.

```
[196]:
# Function to plot line chart of hour in a 24 hour range
def time_line_chart(dataset, col):
    df_pickup = dataset.copy()
    df_pickup[col] = pd.to_datetime(dataset[col], format='%H:%M:%S').dt.time
    df_pickup['Hour'] = pd.to_datetime(df_pickup[col], format='%H:%M:%S').dt.hour

    # Group by Pickup_Hour to get the frequency of pickups by hour
    hourly_pickups = df_pickup.groupby('Hour').size()

    # Plot the line chart
    plt.figure(figsize=(10, 6))
    plt.plot(hourly_pickups.index, hourly_pickups.values, marker='o', color='skyblue', linestyle='-', linewidth=2)
    plt.xlabel('Hour of the Day')
    plt.ylabel('Number of Pickups')
    plt.title(f'{col} Hour Distribution Throughout the Day')
    plt.xticks(range(0, 24))  # Show all hours from 0 to 23
    plt.grid(True)
```

Figure 21

I created a function to facilitate the visualisation of time by hour.

```
time_line_chart(dataset, "Pickup_Time")
```

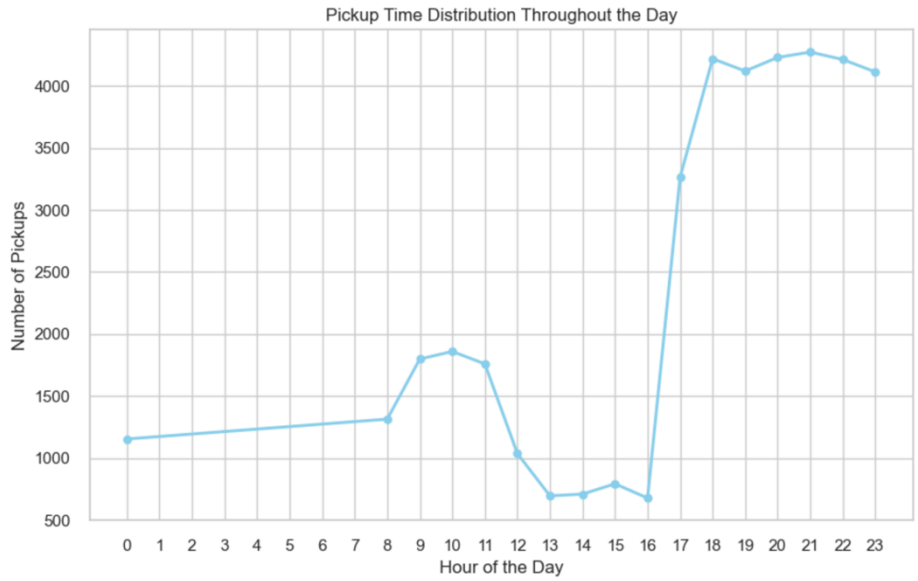


Figure 22

The line chart reveals that there is usually a spike in deliveries to be made between 16:00 and 17:00 which then peaks at 21:00 where it begins to decrease.

```
[133]: plot_frequency(dataset,'Day_of_Week', 'all')
```

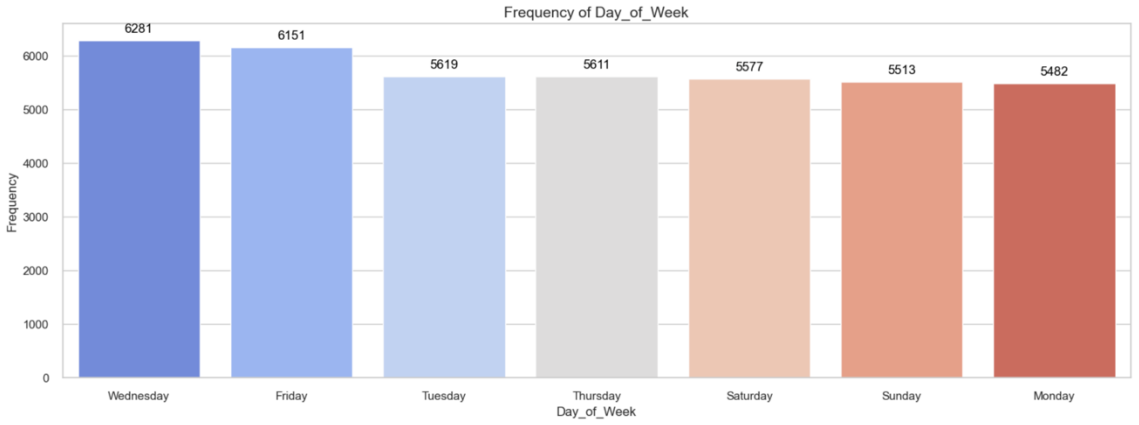


Figure 23

Deliveries for each day of the week are almost evenly distributed but with Wednesday with 6,281 instances as the highest and Monday with 5,482 as the lowest.

From the analysis and visualisations conducted based on time, we can conclude that delivery times are the busiest on Wednesdays between the hours of 17:00 – 23:00.

The next image shows the frequency bar chart plotted against the Area column.

```
[134]: plot_frequency(dataset,'Area', 'all')
```

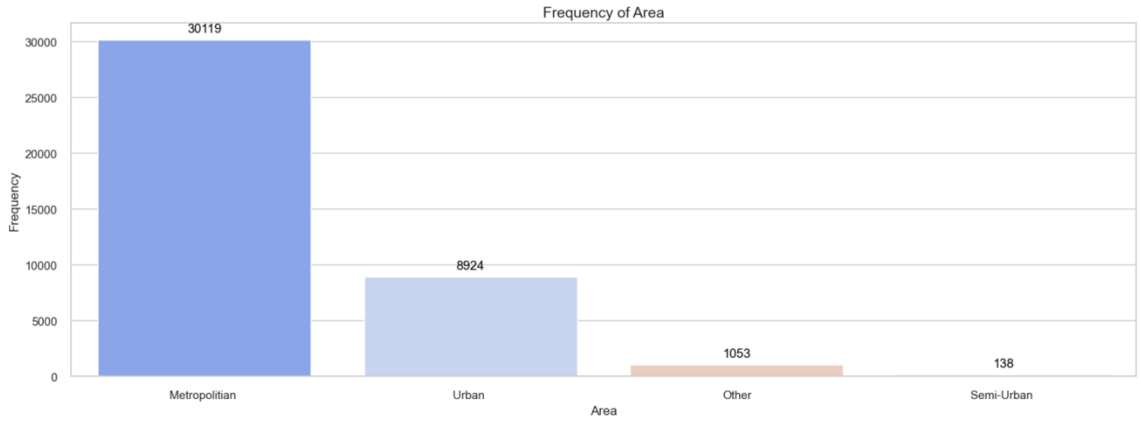


Figure 24

The visualisation reveals that most deliveries are made within the Metropolitan areas and the least at Semi-Urban areas.

4. Cluster Modelling

For modelling this dataset, I have decided to use only a sample of the data to demonstrate the machine learning processes aimed at planning and optimizing deliveries.

```
[138]: #Taking the busiest day plus 13 consecutive days for 2 week's worth of data

busiest_city_busiest_day = busiest_city['Order_Date'].mode()[0]
fourteen_consecutive_days = [busiest_city_busiest_day]
for i in range(1, 14):
    fourteen_consecutive_days.append(busiest_city_busiest_day + pd.Timedelta(days=i))
print(fourteen_consecutive_days)
```

```
[Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:00'), Time
stamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'), Timestam
p('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Timestamp('20
22-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
```

```
[139]: two_weeks_df = busiest_city[busiest_city["Order_Date"].isin(fourteen_consecutive_days)]
```

```
[140]: two_weeks_df.count()
```

```
[140]: Order_ID              1071
       Store_Latitude        1071
       Store_Longitude       1071
       Drop_Latitude         1071
       Drop_Longitude        1071
       Order_Date            1071
       Order_Time            1071
       Pickup_Time           1071
       Weather               1071
       Traffic               1071
       Vehicle               1071
       Area                  1071
       Category              1071
       Time_Category         1071
       Day_of_Week           1071
       Store_City            1071
       Store_Country         1071
       Destination_City      1071
       Destination_Country   1071
       Distance_km           1071
       dtype: int64
```

Figure 25

Two weeks' worth of data has been extracted from the busiest city for modelling. This contains 1,071 instances and will be sufficient to demonstrate the general idea of delivery optimisation using the clustering technique.

```
[141]: plot_map(df=two_weeks_df)
```

[141]:

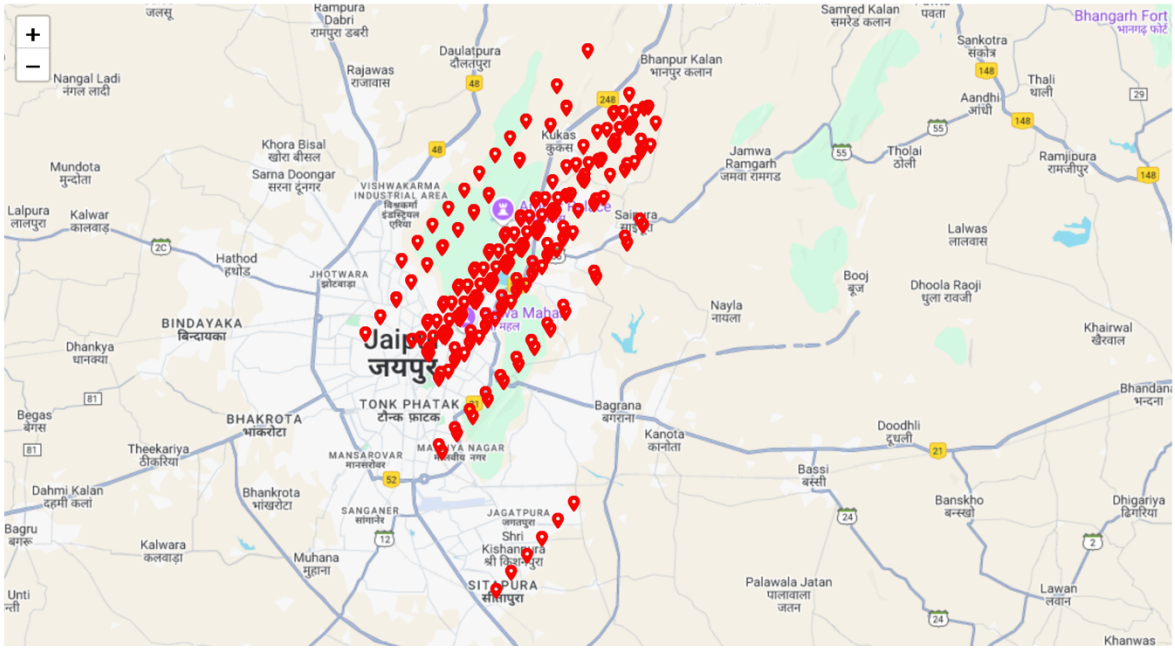


Figure 26

This map view reveals that the instances selected are situated in the Jaipur city.

4.1 Feature selection

```
[146]: # Selecting features for training
focus_training_dataset = two_weeks_df[["Store_Latitude", "Store_Longitude", "Drop_Latitude", "Drop_Longitude",
                                       "Area", "Category", "Pickup_Time", "Time_Category"]]
```

```
[147]: focus_training_dataset.head()
```

[147]:

	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude	Area	Category	Pickup_Time	Time_Category
25	26.849596	75.800512	26.879596	75.830512	Urban	Cosmetics	20:40:00	Night (18:00 - 00:00)
57	26.888420	75.800689	26.898420	75.810689	Other	Shoes	09:40:00	Morning (06:00 - 12:00)
59	26.913987	75.752891	26.993987	75.832891	Urban	Cosmetics	22:35:00	Night (18:00 - 00:00)
117	26.892312	75.806896	26.922313	75.836896	Metropolitian	Jewelry	21:05:00	Night (18:00 - 00:00)
214	26.911927	75.797282	27.051927	75.937282	Metropolitian	Snacks	23:00:00	Night (18:00 - 00:00)

```
[148]: focus_training_dataset.describe()
```

[148]:

	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude
count	1071.000000	1071.000000	1071.000000	1071.000000
mean	26.898673	75.793756	26.965807	75.860890
std	0.031718	0.016907	0.053684	0.042798
min	26.766536	75.752820	26.776536	75.762821
25%	26.892312	75.792934	26.932908	75.827282
50%	26.905287	75.794592	26.963726	75.857333
75%	26.913483	75.802300	27.001378	75.890753
max	26.956431	75.837333	27.086431	75.950753

Figure 27

7 features have been selected for this modelling as shown in figure 11. These features are crucial for our main goal of delivery planning. These included coordinates for store location and drop location and area to cluster based on location proximity, Category to cluster based on goods categories, Pick_up time and Time category to cluster based on Time proximity of deliveries.

4.2 Model Training

[827]:
```python
# CLASS FOR AUTOMATED MODEL DEVELOPMENT

class ModelTrainer:
    def __init__(self, dataframe, priorities):
        # Initialize the ModelTrainer class with the main dataset, priorities, and other essential attributes.
        self.dataset = dataframe
        self.main_dataframe = two_weeks_df
        self.labeled_dataset = None
        self.priorities = priorities
        self.preprocessed = None
        self.labels = dict()
        self.y_kmeans = dict()

        # Define random colors to assist with cluster visualization for different dimensions.
        self.colors = {
            "time": ["#00008B", "#2E8B57", "#FFD700", "#FF8C00", "#FF4500", "#FF6347", "#FF69B4", "#6A5ACD", "#483D8B", "#8B0000"],
            "pickup_location": ["#FF5733", "#33FF57", "#3357FF", "#FFD700", "#900C3F", "#FFC300", "#581845", "#DAF7A6", "#C70039", "#900C3F"],
            "store_location": ["#FF5733", "#33FF57", "#3357FF", "#FFD700", "#900C3F", "#FFC300", "#581845", "#DAF7A6", "#C70039", "#900C3F"],
            "goods_Category": ["#FF6347", "#3CB371", "#4682B4", "#FFD700", "#6A5ACD", "#FF1493", "#40E0D0", "#FFA500", "#DC143C", "#ADFF2F"]
        }
        print("ModelTrainer Instance created successfully")
```

Figure 28

The ModelTrainer class is a tool designed to streamline the training and evaluation of clustering models, focusing on organizing delivery data into meaningful clusters. It allows flexible experimentation with various clustering methods and configurations, supporting optimized delivery planning.

Method	Description
__init__()	Initializes the instance with dataset, clustering priorities, and structures for storing cluster labels.
numeric_encoder()	One-hot encodes categorical variables Area, Category, Time_Category).
cyclical_encoder()	Encodes Pickup_Time as cyclical features (hour_sin, hour_cos) for time patterns using cosine transformations
data_scaler()	Scales latitude and longitude coordinates using MinMax scaling.
raw_preprocess()	Combines encoded, cyclical, and scaled features into a single DataFrame for clustering.
get_encoded_array()	Converts the preprocessed DataFrame to a NumPy array for training.
weighted_array()	Adjusts feature weights based on clustering priorities.
view_elbow()	Plots an elbow curve to find the best cluster number (k) for a given dimension.
view_elbow_alt()	Alternative elbow plot using KElbowVisualizer.

view_dendrogram()	Shows a dendrogram for hierarchical clustering to determine k.
get_silhouette_score()	Calculates silhouette scores to find the optimal k.
get_pca()	Applies PCA to reduce data to 2D for visualization, displaying explained variance.
train_KMeans_algorithm()	Trains a KMeans model based on priorities, using k from silhouette scores or predefined values.
train_Agglo_algorithm()	Trains Agglomerative Clustering, assigning cluster labels to each dimension.
label_main_dataframe()	Adds cluster labels and colours to the main DataFrame for visualization.
view_scatter_plot()	Plots clusters in 2D PCA-reduced space with color-coded clusters.
map_view()	Filters data by delivery plan and displays it on a map for geographic analysis.

4.2.1 ModelTrainer Instantiation

[828]:
```
# Setting feature groups and priorities
feature_group = {
    "time": ["hour_sin", "hour_cos", "Time_Category"],
    "pickup_location": ["Drop_Latitude", "Drop_Longitude"],
    "store_location": ["Store_Latitude", "Store_Longitude"],
    "goods_Category": ["Category"],
}
priorities = {
    "time": {"features":feature_group["time"], "importance": 2, "clusters": "auto"},
    "pickup_location": {"features":feature_group["pickup_location"], "importance": 10, "clusters": "auto"},
    "goods_Category": {"features":feature_group["goods_Category"], "importance": 4, "clusters": "auto"}
}
```

[829]:
```
# Creating an instance of the model trainer
DeliveryModel = ModelTrainer(dataframe=focus_training_dataset, priorities=[priorities, feature_group])
```

```
ModelTrainer Instance created successfully
```

Figure 29

Before initiating the ModelTrainer, predefined feature groups and clustering priorities guide how the model creates clusters within the generated objects.
Feature Groups:

- **Time**: Includes hour_sin, hour_cos, and Time_Category to capture temporal patterns in deliveries.
- **Pickup Location**: Comprises Drop_Latitude and Drop_Longitude for geographic clustering.
- **Store Location**: Contains Store_Latitude and Store_Longitude.
- **Goods Category**: Includes Category to classify product types.

Clustering Priorities:

- **Time**: Importance = 2, Clusters = Auto
- **Pickup Location**: Importance = 10, Clusters = Auto
- **Goods Category**: Importance = 4, Clusters = Auto

The model assigns importance to each feature in a group, prioritizing it during modelling. For example, "time" is assigned to hour_sin, hour_cos, and Time_Category features. The model uses the silhouette score to determine the optimal number of clusters, which can be adjusted based on new insights.

4.2.2 Data Preprocessing

```python
def numeric_encoder(self):
    # One-hot encode categorical columns: 'Area', 'Category', 'Time_Category'.
    encode = OneHotEncoder(drop='first')
    columns = ['Area', 'Category', "Time_Category"]

    # Fit and transform the categorical data.
    encoded_array = encode.fit_transform(self.dataset[columns]).toarray()
    encoded = pd.DataFrame(encoded_array, columns=encode.get_feature_names_out(columns))
    return encoded

def cyclical_encoder(self):
    # Encode time data cyclically using sine and cosine transformations.
    time_dataset = self.dataset[['Pickup_Time']].copy()

    # Extract the hour from the Pickup_Time and apply cyclic encoding.
    time_dataset['hour'] = pd.to_datetime(time_dataset['Pickup_Time'], format='%H:%M:%S').dt.hour
    columns = ['hour_sin', 'hour_cos']
    time_dataset[columns[0]] = np.sin(2 * np.pi * time_dataset['hour'] / 24)
    time_dataset[columns[1]] = np.cos(2 * np.pi * time_dataset['hour'] / 24)

    # Convert to DataFrame for consistency with other features.
    cyclical_features = time_dataset[columns].to_numpy()
    encoded_cyclical_features = pd.DataFrame(cyclical_features, columns=columns)
    return encoded_cyclical_features

def data_scaler(self):
    # Scale latitude and longitude coordinates between 0 and 1 for clustering.
    scaler = MinMaxScaler()
    columns = ['Store_Latitude', 'Store_Longitude', 'Drop_Latitude', 'Drop_Longitude']

    # Apply MinMax scaling to ensure coordinates are in a comparable range.
    features = scaler.fit_transform(self.dataset[columns])
    df = pd.DataFrame(features, columns=columns)
    return df
```

Figure 30

```python
def raw_preprocess(self):
    # Combine all preprocessed features into a final DataFrame.
    non_numerical_features = self.numeric_encoder()  # Categorical features
    cyclical_features = self.cyclical_encoder()      # Time-based features
    numerical_features = self.data_scaler()          # Scaled spatial features

    # Concatenate all features to create the preprocessed dataset.
    final_preprocessed = pd.concat([numerical_features, cyclical_features, non_numerical_features], axis=1)
    self.preprocessed = final_preprocessed
    return final_preprocessed
```

Figure 31

```
[830]: DeliveryModel.raw_preprocess().head(3)
```

[830]:

	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude	hour_sin	hour_cos	Area_Other	Area_Semi-Urban	Area_Urban	Category_Books	...	Category_C
0	0.437400	0.564316	0.332564	0.360189	-0.866025	0.500000	0.0	0.0	1.0	0.0	...	
1	0.641849	0.566410	0.393307	0.254709	0.707107	-0.707107	1.0	0.0	0.0	0.0	...	
2	0.776487	0.000840	0.701693	0.372848	-0.500000	0.866025	0.0	0.0	1.0	0.0	...	

3 rows × 27 columns

Figure 32

The raw_preprocess method concatenates data retrieved from the numeric_encoder, cyclical_encoder and data_scaler methods.

- numeric_encoder encodes categorical data using OneHotEncoder to convert them to a numeric binary format and returns a dataframe of the encoded data.

- cyclical_encoder encodes the pick_up time by extracting the hour component and applying a cosine transformation to the hour to represent the cyclical nature of hours of the day and to preserve its continuity.
- data_scaler performs a MinMax scaler to the numeric features which reduces them to a value between minimum of 0 and maximum of 1.

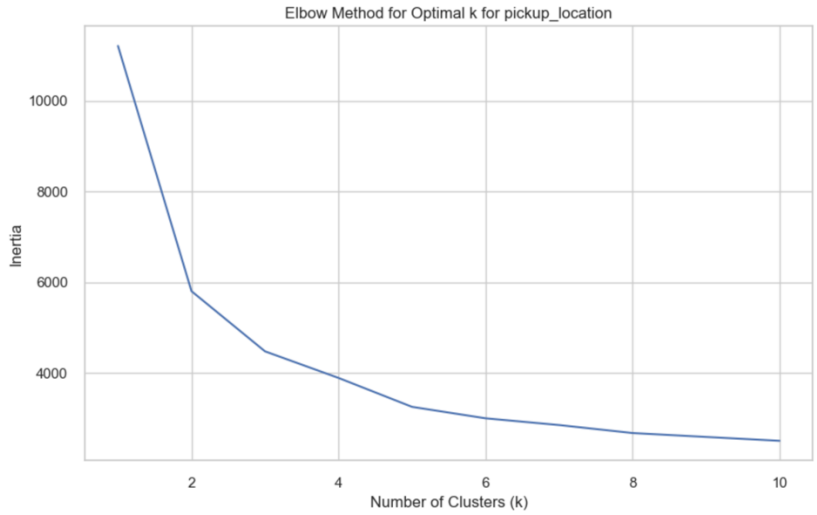
The final output in figure 32 shows a dataframe which was preprocessed using a combination of these encoders and scaler.

4.2.2 Optimal K visualisation using elbow method

```
def get_encoded_array(self):
    # Get preprocessed data as a NumPy array for clustering.
    return self.raw_preprocess().to_numpy()

def weighted_array(self, dimension):
    # Adjust feature importance based on specified priorities for clustering.
    features_list = self.priorities[1][dimension]  # Priority features for the given dimension
    enc_array = self.get_encoded_array()

    # Apply weights to specified features based on priorities.
    for index, column in enumerate(self.preprocessed.columns):
        for f in features_list:
            if f in column:
                enc_array[:, index] *= self.priorities[0][dimension]["importance"]
    return enc_array

def view_elbow(self, dimension):
    # Visualize the elbow method for optimal k selection.
    inertia = []
    for i in range(1, 11):
        kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
        if dimension:
            kmeans.fit(self.weighted_array(dimension))
        else:
            kmeans.fit(self.get_encoded_array())
        inertia.append(kmeans.inertia_)

    # Plot the inertia (within-cluster sum of squares) for different values of k.
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, 11), inertia)
    plt.title(f'Elbow Method for Optimal k for {dimension}')
    plt.xlabel('Number of Clusters (k)')
    plt.ylabel('Inertia')
    plt.show()
```

Figure 33

```
[831]: # Viewing elbow when location is prioritised

DeliveryModel.view_elbow("pickup_location")
```



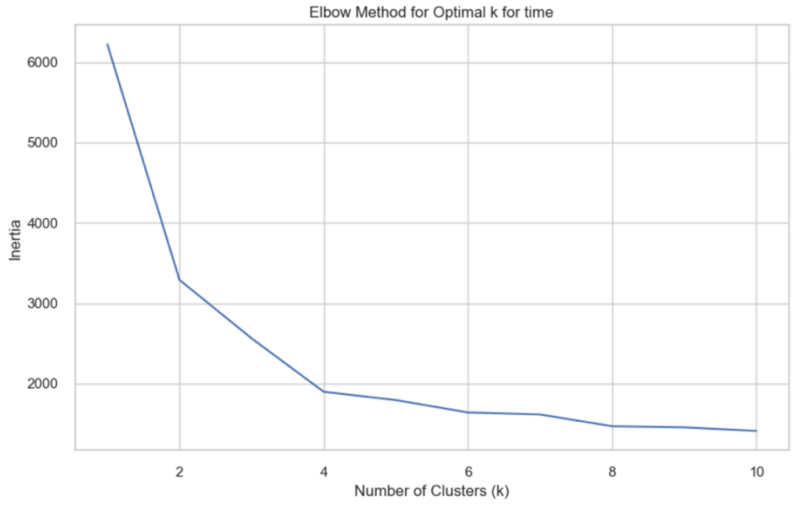
The code and visualization suggest that the optimal K is 5 at the 3rd elbow when the pickup location is prioritized. The view_elbow method plots the elbow chart using a weighted array or a uniformly weighted array.

```
[832]: # Resetting pickup_location clusters

DeliveryModel.priorities[0]["pickup_location"]["clusters"] = 5
print(DeliveryModel.priorities[0]["pickup_location"])
```

```
{'features': ['Drop_Latitude', 'Drop_Longitude'], 'importance': 10, 'clusters': 5}
```

Resetting K for pick_up location cluster to 5 as determined from the elbow.

[833]:
```
# Viewing elbow when time is prioritised

DeliveryModel.view_elbow("time")
```



[834]:
```
# Resetting time clusters

DeliveryModel.priorities[0]["time"]["clusters"] = 4
print(DeliveryModel.priorities[0]["time"])
```

```
{'features': ['hour_sin', 'hour_cos', 'Time_Category'], 'importance': 2, 'clusters': 4}
```

Checking and resetting for K to 4 when time dimension is prioritised

Using KMeans Algorithm

```
def train_KMeans_algorithm(self):
    # Train KMeans clustering algorithm based on specified priorities or automatically chosen k.
    print("Using KMeans Algorithm")

    if len(self.priorities[0]) > 0:
        for i, feature_ in enumerate(self.priorities[0]):
            k = self.priorities[0][feature_]["clusters"]
            if k == "auto":
                k = self.get_silhouette_score(feature_)

            kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
            clusters = kmeans.fit(self.weighted_array(feature_))

            self.labels[feature_] = clusters.labels_
            self.y_kmeans[feature_] = clusters.predict(self.weighted_array(feature_))
            print(f"{feature_} cluster group created  with k={k}: {sorted(list(set(self.labels[feature_])))}")
    else:
        k = self.get_silhouette_score(None)
        kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
        clusters = kmeans.fit(self.get_encoded_array())
        self.labels[feature_] = clusters.labels_
        self.y_kmeans[feature_] = clusters.predict(self.get_encoded_array())
        print(f"{feature_} cluster group created  with k={k}: {sorted(list(set(self.labels[feature_])))}")
```

The code in the figure demonstrates a model trainer class method for training using the KMeans algorithm, which updates priority groups by adding weights to each specified feature before applying the algorithm.

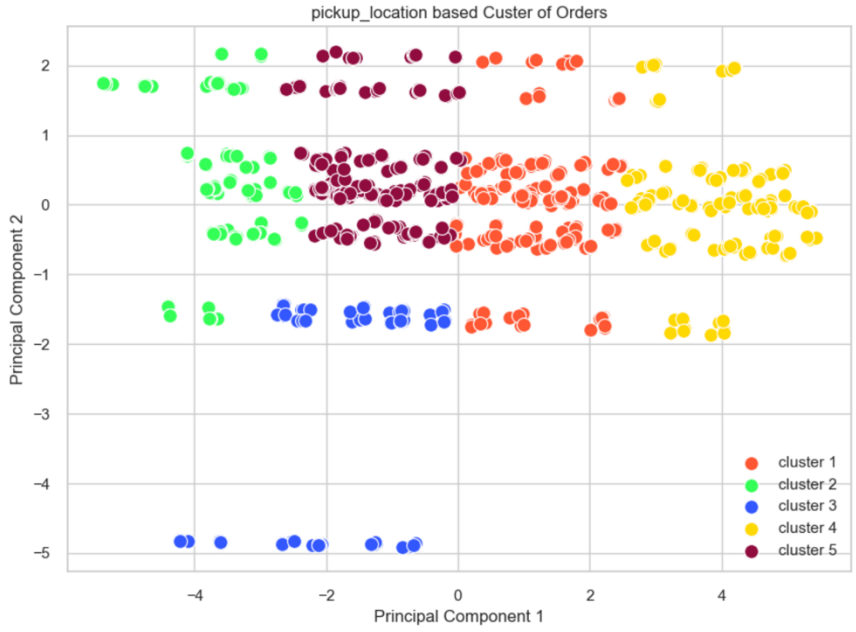
[73]:
```
# Training model using KMeans Algorithm

DeliveryModel.train_KMeans_algorithm()
```

```
Using KMeans Algorithm
time cluster group created  with k=4: [0, 1, 2, 3]
pickup_location cluster group created  with k=5: [0, 1, 2, 3, 4]

evaluated k=10 for goods_Category
goods_Category cluster group created  with k=10: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The clusters have been created for each cluster group: Time cluster, pickup_location cluster and goods_category cluster. Time and location k number of clusters were set manually with the aid of the elbow method. Goods category k clusters was computed automatically using the silhouette score which was computed as 10 clusters.

```
[75]: # Viewing scatter plot for location clusters

DeliveryModel.view_scatter_plot("pickup_location")
```

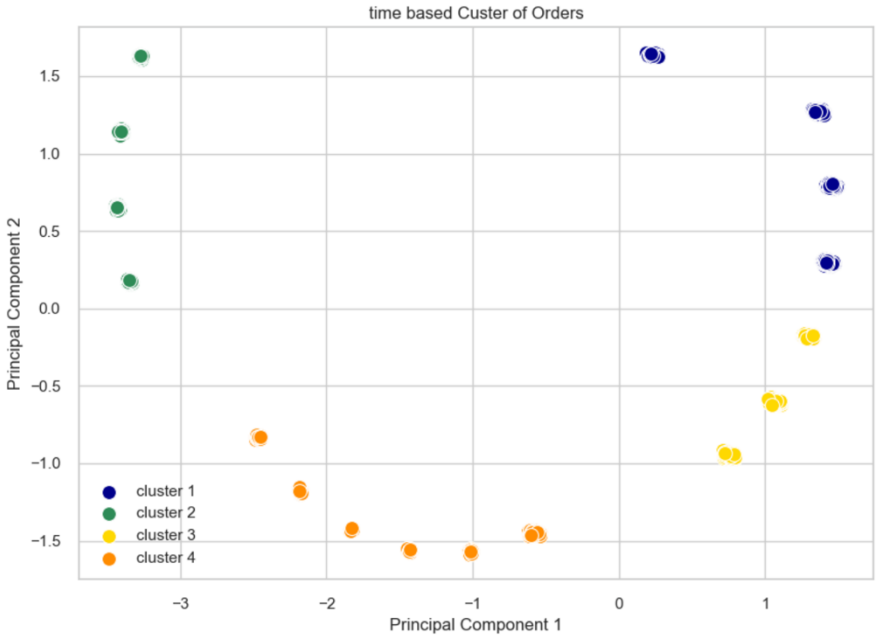
```
explained variance ratio: 0.8185547545039022
```



The scatter plot has been visualised to view the cluster groups for pick up location, different pick-up locations are represented as datapoints in the scatter plot.

```
[76]: # Viewing scatter plot for time clusters

DeliveryModel.view_scatter_plot("time")
```

```
explained variance ratio: 0.7188040215906443
```



The time clusters has been visualised and we can already see that it is almost cyclical meaning, mimicking the natural cyclical pattern of the hours of the day.

```
[77]: DeliveryModel.label_main_dataframe().head(3)
```

[77]:

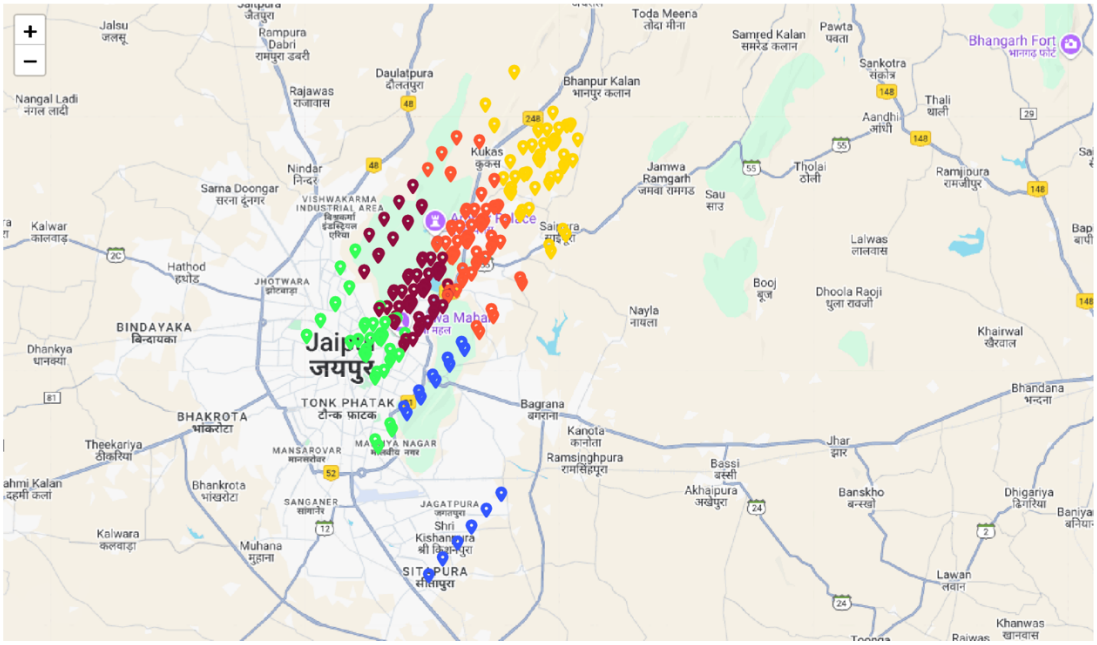
Distance_km	time_Cluster	time_Cluster_Color	pickup_location_Cluster	pickup_location_Cluster_Color	goods_Category_Cluster	goods_Category_Cluster_Color
4.470286	3	#FFD700	3	#3357FF	8	#FFA500
1.489927	2	#2E8B57	2	#33FF57	3	#4682B4
11.916583	1	#00008B	5	#900C3F	8	#FFA500

The clusters and respective colors have been concatenated to the dataframe that was used for clustering.

Analysis of clusters

```
[79]: planner = {
          "Order_Date": fourteen_consecutive_days[0:14], # Change index to view for individual days or a group of days
          "pickup_location_Cluster": [1, 2, 3, 4, 5],  # Change index to view for individual cluster or a group of clusters (1, 2, 3, 4, 5)
          "time_Cluster": [1, 2, 3, 4],
          "color": "pickup_location_Cluster"
      }

      DeliveryModel.map_view(plan=planner)
```

```
Dates: [Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:0
0'), Timestamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'),
Timestamp('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Times
tamp('2022-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
Count: 1071
```



I have created a planner which takes in a cluster or a group of listers and visualises them on the map. The cluster in the image above contains time clusters for 1- 4 and location clusters of 1-5. The ‘color’ key is set to pickup_location_Cluster to visualise the cluster colors based on location groupings. It can be set time_Cluster to visualise the time cluster grouping.

The planner can be edited to view specific clusters, this can be useful for assigning cluster groups to a particular driver .

```
[81]: planner = {
          "Order_Date": fourteen_consecutive_days[0:14], # Change index to view for individual days or a group of days
          "pickup_location_Cluster": [1, 2, 3, 4, 5],  # Change index to view for individual cluster or a group of clusters (1, 2, 3, 4, 5)
          "time_Cluster": [1, 2, 3, 4],
          "color": "time_Cluster"
      }

      DeliveryModel.map_view(plan=planner)

      Dates: [Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:0
      0'), Timestamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'),
      Timestamp('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Times
      tamp('2022-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
      Count: 1071
```

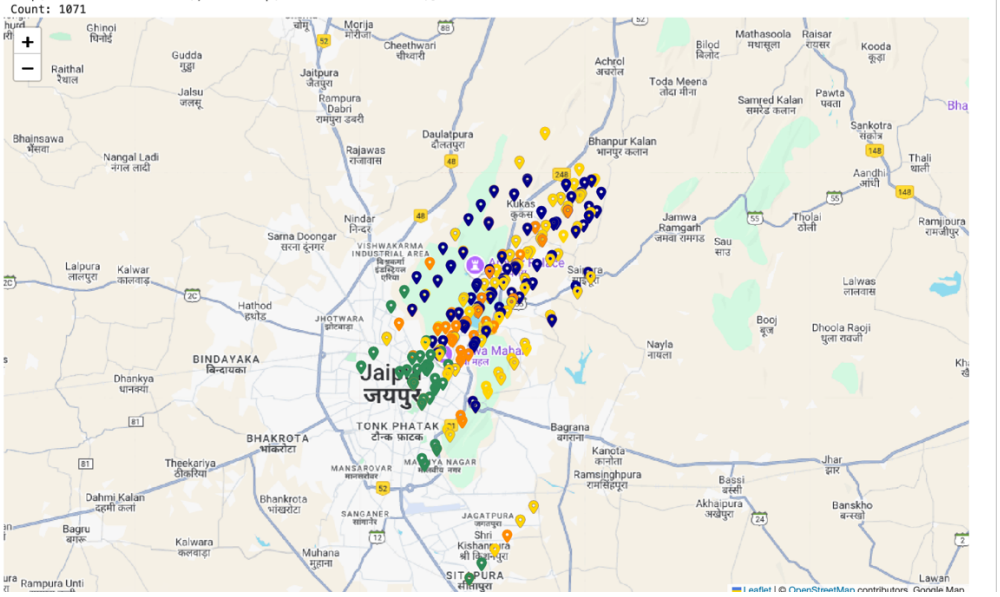
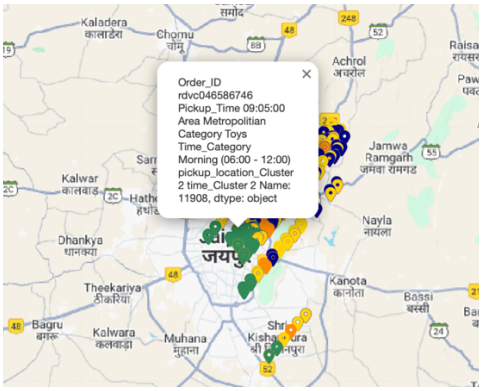
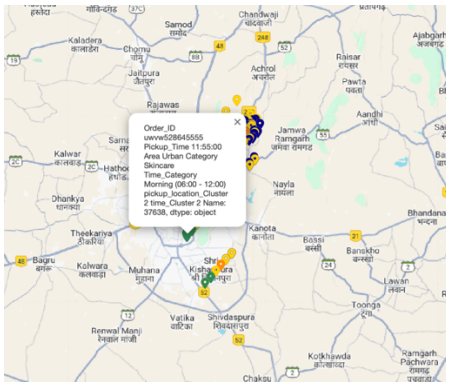
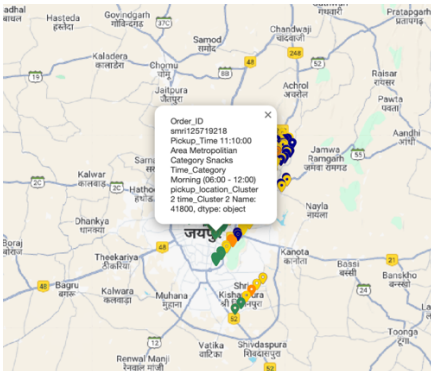
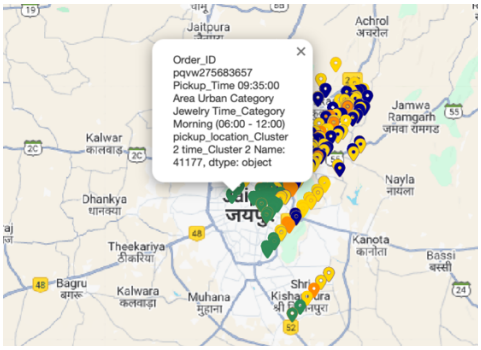


Figure 34: Viewing clusters based on time

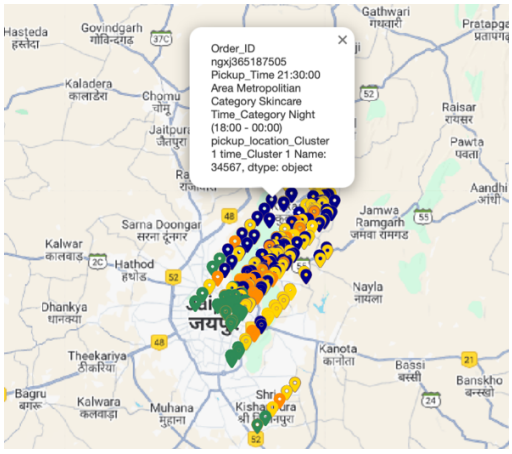


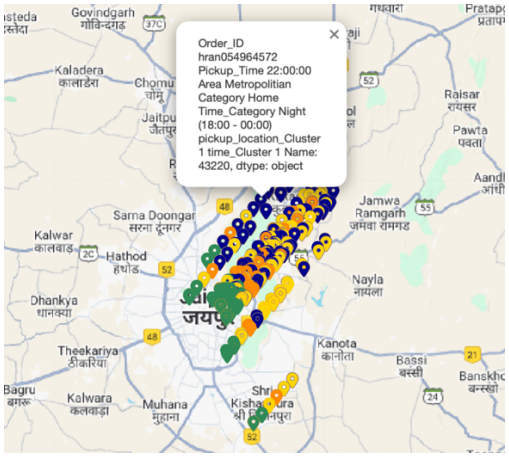






We can see from the 4 different datapoint in same time cluster (2) exhibit same range of time (Morning 6am – 12pm) and the actual pick-up times are close together which means that one driver can handle deliveries for this time cluster in same location cluster.





Inspecting a different time cluster group (cluster 1) shows that the deliveries in these clusters are for nighttime within 18:00 – 00:00 and the time are close enough.

AGGLOMERATIVE CLUSTERING

ALGO CLUSTERING

```
[82]: AggloDeliveryModel = ModelTrainer(dataframe=focus_training_dataset, priorities=[priorities, feature_group])
      ModelTrainer Instance created successfully

[83]: AggloDeliveryModel.raw_preprocess()
```

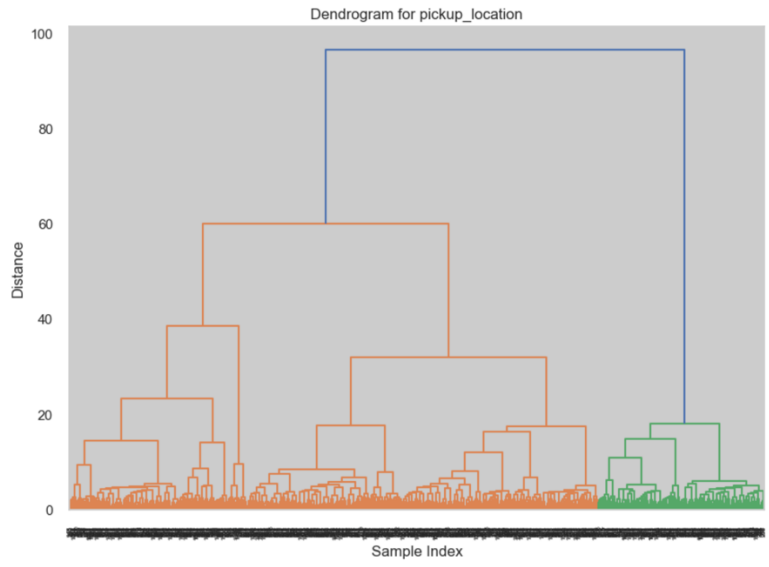
[83]:

	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_Longitude	hour_sin	hour_cos	Area_Other	Area_Semi-Urban	Area_Urban	Category_Books	...	C
0	0.437400	0.564316	0.332564	0.360189	-0.866025	5.000000e-01	0.0	0.0	1.0	0.0	...	
1	0.641849	0.566410	0.393307	0.254709	0.707107	-7.071068e-01	1.0	0.0	0.0	0.0	...	
2	0.776487	0.000840	0.701693	0.372848	-0.500000	8.660254e-01	0.0	0.0	1.0	0.0	...	
3	0.662345	0.639854	0.470408	0.394159	-0.707107	7.071068e-01	0.0	0.0	0.0	0.0	...	
4	0.765639	0.526097	0.888659	0.928320	-0.258819	9.659258e-01	0.0	0.0	0.0	0.0	...	
...	...	...	...	...	...	...	...	...	...	...	...	...
1066	0.765639	0.526097	0.565969	0.396212	-0.707107	-7.071068e-01	0.0	0.0	1.0	0.0	...	
1067	0.437400	0.564316	0.397102	0.466610	-0.258819	9.659258e-01	0.0	0.0	0.0	0.0	...	
1068	0.656442	0.582904	0.466787	0.368548	-1.000000	-1.836970e-16	0.0	0.0	0.0	0.0	...	
1069	0.775113	0.000000	0.507236	0.053205	0.258819	-9.659258e-01	0.0	0.0	0.0	0.0	...	
1070	0.718313	0.475513	0.440162	0.213833	0.500000	-8.660254e-01	0.0	0.0	1.0	0.0	...	

1071 rows × 27 columns

The preprocessed data is gotten by the raw_preprocess method from the model Trainer class and is ready for modelling.

```
[84]: AggloDeliveryModel.view_dendrogram("pickup_location")
```

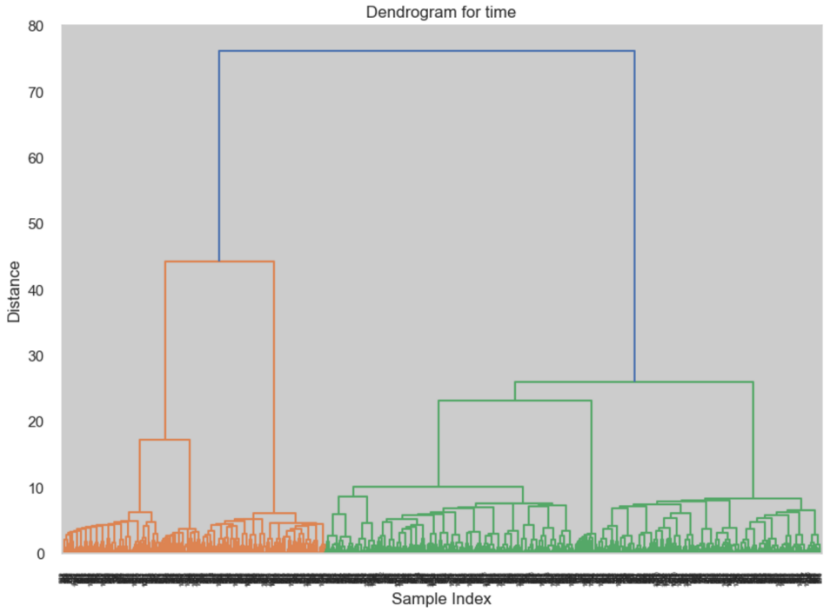



```
[93]: # Resetting pickup_location clusters

      AggloDeliveryModel.priorities[0]["pickup_location"]["clusters"] = 3
      print(AggloDeliveryModel.priorities[0]["pickup_location"])

      {'features': ['Drop_Latitude', 'Drop_Longitude'], 'importance': 10, 'clusters': 3}
```

Figure 35: Dendrogram for location cluster

The dendrogram is plotted to get k as 3 for location cluster when cutting a line at the middle of the graph.

```
[94]: AggloDeliveryModel.view_dendrogram("time")
```




```
[95]: # Resetting pickup_location clusters

AggloDeliveryModel.priorities[0]["time"]["clusters"] = 3
print(AggloDeliveryModel.priorities[0]["time"])
```

```
{'features': ['hour_sin', 'hour_cos', 'Time_Category'], 'importance': 2, 'clusters': 3}
```

Figure 36: Dendrogram for time cluster

K cluster for time category as 3 when cutting from the midpoint of the dendrogram.

```
def train_Agglo_algorithm(self):
    # Train Agglomerative Clustering based on priorities or automatically chosen k.
    print("Using AgglomerativeClustering Algorithm")

    if len(self.priorities[0]) > 0:
        for i, feature_ in enumerate(self.priorities[0]):
            k = self.priorities[0][feature_]["clusters"]
            if k == "auto":
                k = self.get_silhouette_score(feature_)

            agg_clustering = AgglomerativeClustering(n_clusters=k, metric='euclidean', linkage='ward')
            labels = agg_clustering.fit_predict(self.weighted_array(feature_))

            self.labels[feature_] = labels
            print(f"{feature_} cluster group created  with k={k}: {sorted(list(set(labels)))}")
    else:
        k = self.get_silhouette_score(None)
        agg_clustering = AgglomerativeClustering(n_clusters=k, metric='euclidean', linkage='ward')
        labels = agg_clustering.fit_predict(self.get_encoded_array())
        self.labels[feature_] = labels
        print(f"{feature_} cluster group created  with k={k}: {sorted(list(set(labels)))}")
```

Figure 37: Class method for agglomerative clustering

```
[96]: # Training model using KMeans Algorithm

AggloDeliveryModel.train_Agglo_algorithm()
```

```
Using AgglomerativeClustering Algorithm
time cluster group created  with k=3: [0, 1, 2]
pickup_location cluster group created  with k=3: [0, 1, 2]

evaluated k=9 for goods_Category
goods_Category cluster group created  with k=9: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Creating clusters for the three cluster groups using the AgglomerativeClustering model. Time cluster as 3, location as 3 and category auto computed as 9.

```
[97]: # Viewing scatter plot for location clusters

AggloDeliveryModel.view_scatter_plot("pickup_location")
```

```
explained variance ratio: 0.8185547545039022
```

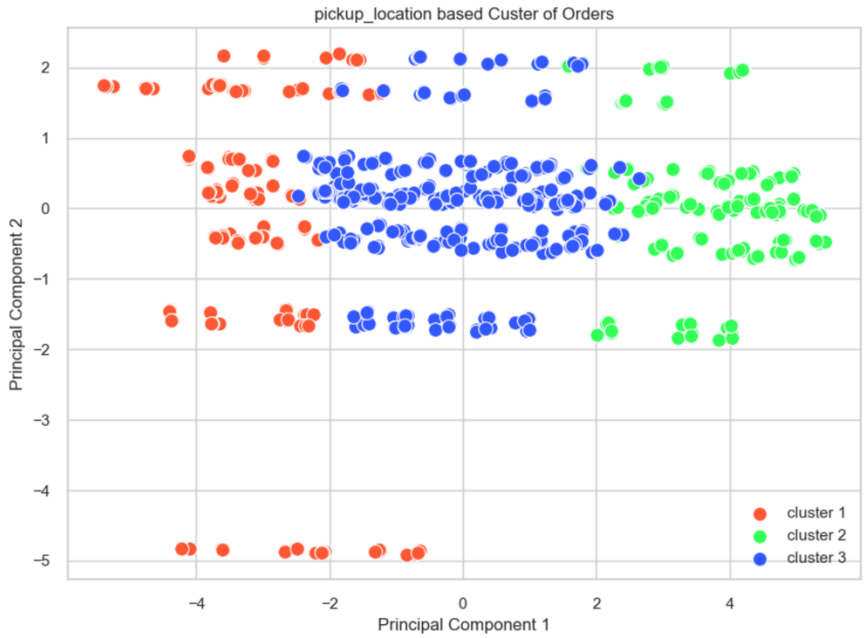


Figure 38: Location Cluster scatterplot for agglomerative clustering.

```
[98]: AggloDeliveryModel.view_scatter_plot("time")
```

```
explained variance ratio: 0.7188040215906443
```

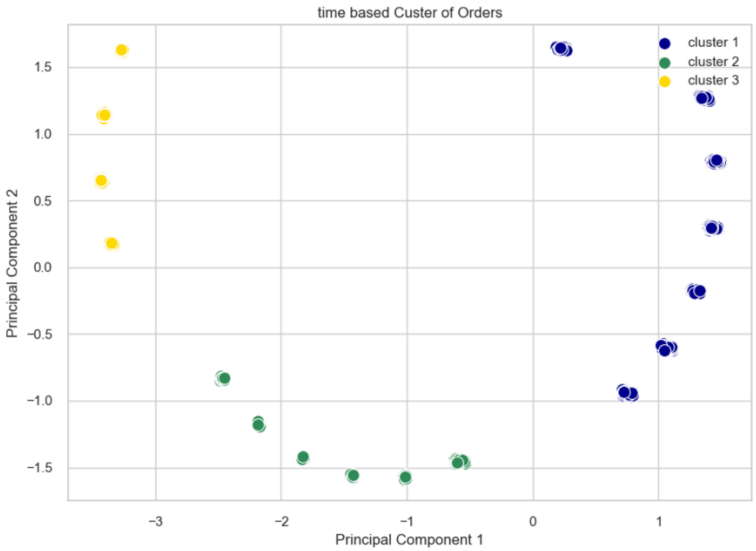


Figure 39: Time clustering scatter plot for Agglomerative clustering

```
[99]: AggloDeliveryModel.label_main_dataframe().head(3)
```

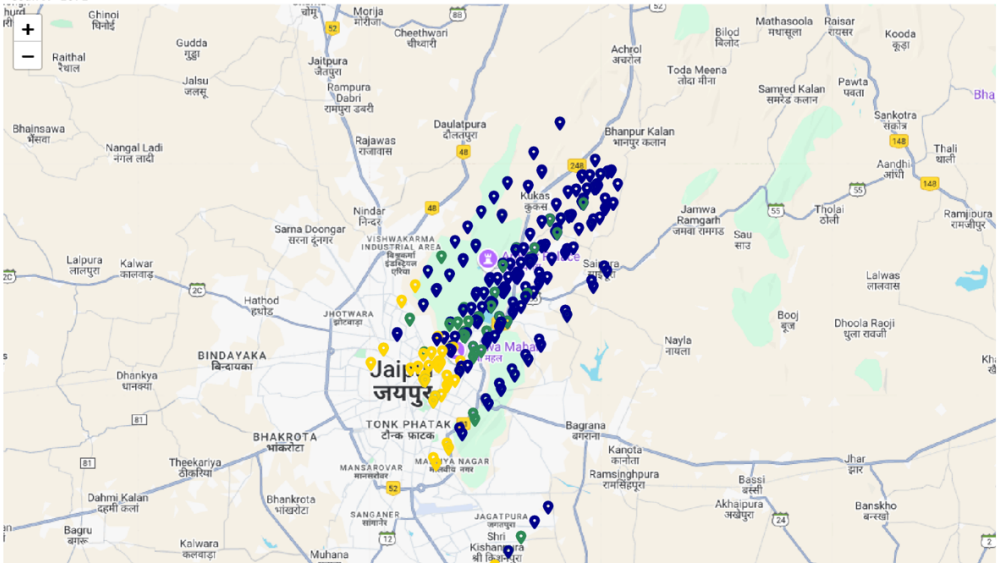
[99]:

Distance_km	time_Cluster	time_Cluster_Color	pickup_location_Cluster	pickup_location_Cluster_Color	goods_Category_Cluster	goods_Category_Cluster_Color
4.470286	1	#00008B	1	#FF5733	1	#FF6347
1.489927	3	#FFD700	1	#FF5733	6	#FF1493
11.916583	1	#00008B	3	#3357FF	1	#FF6347

Cluster number and color has been added the clustering dataframe.

```
[134]: planner = {
           "Order_Date": fourteen_consecutive_days[0:14], # Change index to view for individual days or a group of days
           "pickup_location_Cluster": [1, 2, 3],  # Change index to view for individual cluster or a group of clusters (1, 2, 3, 4, 5)
           "time_Cluster": [1, 2, 3],
           "color": "time_Cluster"
       }

       AggloDeliveryModel.map_view(plan=planner)
```

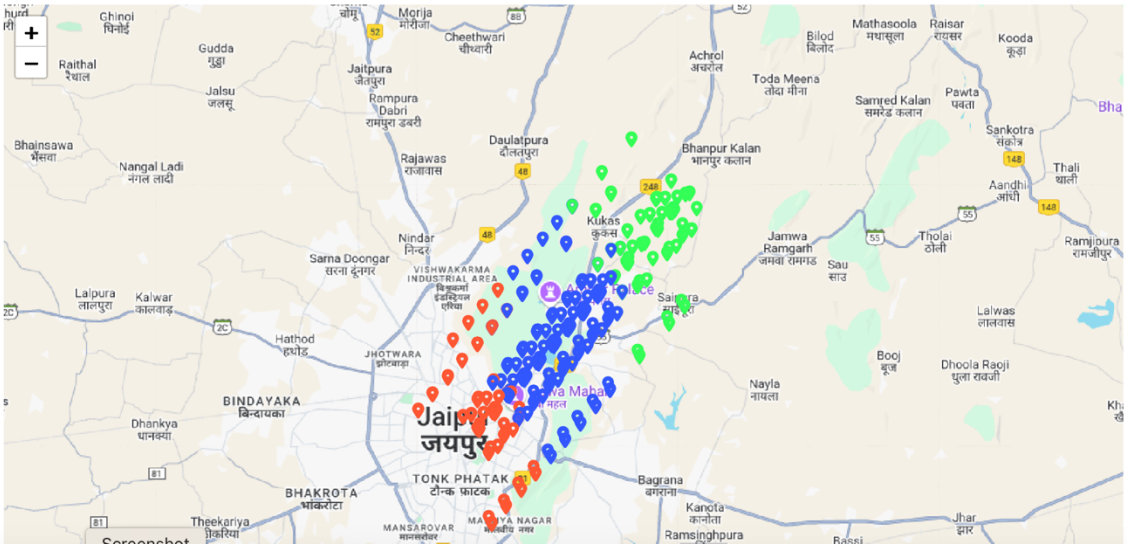
```
Dates: [Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:0
0'), Timestamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'),
Timestamp('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Times
tamp('2022-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
Count: 1071
```



This shows all the clusters with time cluster color

[135]:
```
planner = {
    "Order_Date": fourteen_consecutive_days[0:14], # Change index to view for individual days or a group of days
    "pickup_location_Cluster": [1, 2, 3],  # Change index to view for individual cluster or a group of clusters (1, 2, 3, 4, 5)
    "time_Cluster": [1, 2, 3],
    "color": "pickup_location_Cluster"
}


AggloDeliveryModel.map_view(plan=planner)
```

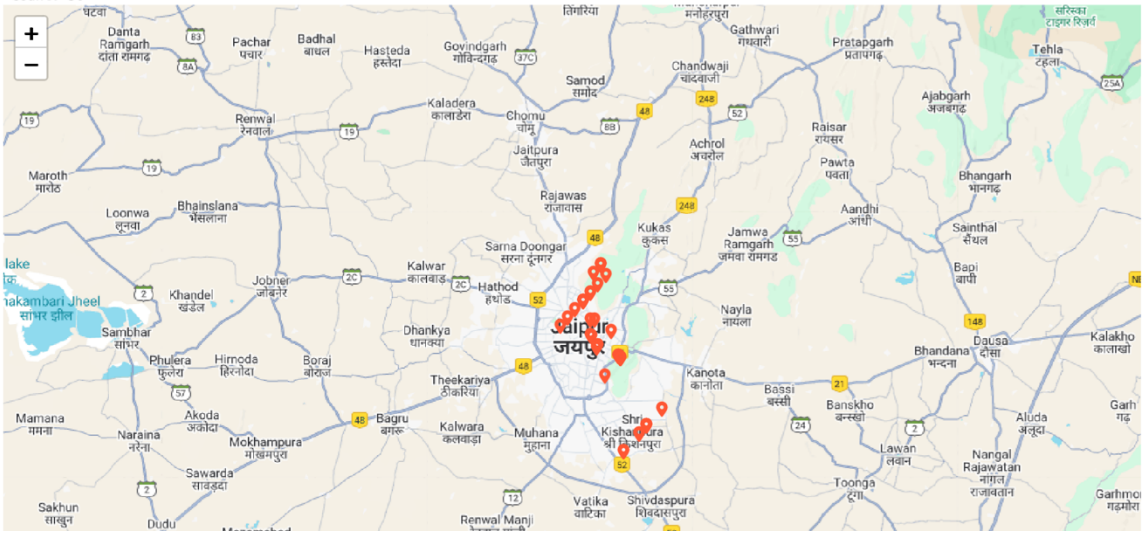
```
Dates: [Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:0
0'), Timestamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'),
Timestamp('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Times
tamp('2022-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
Count: 1071
```



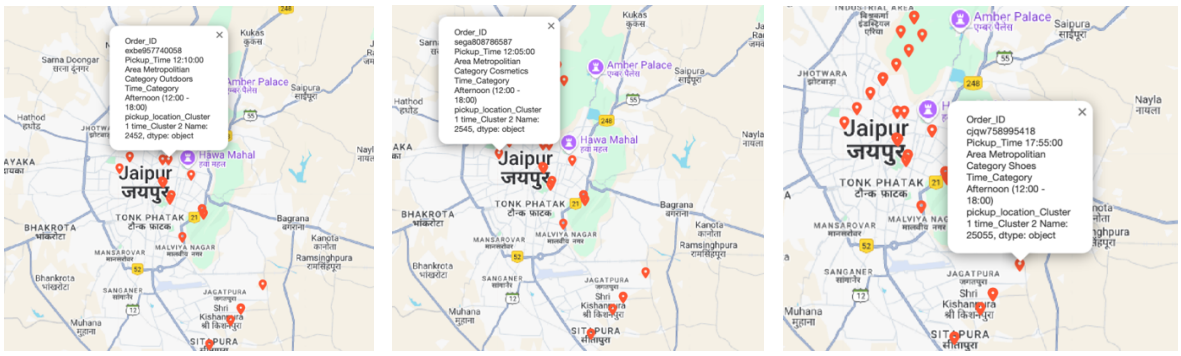
Clusters showing location clusters.

[140]:
```
planner = {
    "Order_Date": fourteen_consecutive_days[0:14], # Change index to view for individual days or a group of days
    "pickup_location_Cluster": [1],  # Change index to view for individual cluster or a group of clusters (1, 2, 3, 4, 5)
    "time_Cluster": [2],
    "color": "pickup_location_Cluster"
}


AggloDeliveryModel.map_view(plan=planner)
```

```
Dates: [Timestamp('2022-03-26 00:00:00'), Timestamp('2022-03-27 00:00:00'), Timestamp('2022-03-28 00:00:00'), Timestamp('2022-03-29 00:00:0
0'), Timestamp('2022-03-30 00:00:00'), Timestamp('2022-03-31 00:00:00'), Timestamp('2022-04-01 00:00:00'), Timestamp('2022-04-02 00:00:00'),
Timestamp('2022-04-03 00:00:00'), Timestamp('2022-04-04 00:00:00'), Timestamp('2022-04-05 00:00:00'), Timestamp('2022-04-06 00:00:00'), Time
tamp('2022-04-07 00:00:00'), Timestamp('2022-04-08 00:00:00')]
Count: 50
```



Visualising the results using the planner dictionary. Location cluster 1 in time cluster 2 shows the few data points with delivery time category around Afternoon (12:00 – 18:00)



Viewing more datapoints inside the time cluster 2 shows all time ranges within afternoon.

This planner can be used to share clusters to different drivers based on time and location proximity. For example, the planner configurations in the figure above can be assigned to one driver.

Conclusion

The results from this clustering task should be self-explanatory to a degree, although here are some actionable insights:

Benefits to the Business

- **Optimized Resource Allocation**: Efficiently assigns vehicles and drivers, reducing redundancy.
- **Cost Reduction**: Minimizes travel distance and fuel costs.
- **Improved Customer Satisfaction**: Ensures timely, reliable deliveries.
- **Enhanced Decision-Making**: Identifies demand patterns for strategic planning.

Actionable Recommendations

- **Optimize Inventory**: Place goods in warehouses closer to demand zones.
- **Dynamic Pricing**: Implement tiered delivery pricing based on urgency.
- **Scale Operations**: Expand hubs and hire staff in high-demand areas.
- **Eco-Friendly Initiatives**: Reduce carbon footprint with optimized routes.
- **Streamline Logistics**: Enhance this model further by using clustering with dynamic routing to handle disruptions.