CT7201: Python Notebooks and Scripting

ANALYSIS OF AMAZON DELIVERY OPERATIONS: FACTORS AFFECTING DELIVERY EFFICIENCY

GROUP E

Table of contents

1. Introduction:

The new era of technology has changed e-commerce to new prospects of business operation and optimised based on customer needs. Amazon, the world's leading online retail industry, is at the vanguard of this change. Amazon gives importance to customer satisfaction which is largely revealed through its products and goods deliverable despite geographical constraints like environmental obstacles or spikes of demand. Delivery operations oversee the transportation of millions of products throughout metropolitan and rural environments, forming the backbone of the logistic system. This report uses an extensive dataset to identify important insights about agent performance, delivery efficiency, and other external factors like traffic and weather, where data-driven insights are investigated into intricate dynamics that impact Amazon's delivery efficiency.

1.1 About Amazon delivery operations

The delivery operations of Amazon represent a complex logistic system globally. It involves order distribution and dispatch, where the Amazon system allocates the package to a nearby delivery centre based on the vicinity of customers, and then agents deliver them to destined places. Evaluating delivery trips, where the journey is influenced by various factors like weather, traffic, and vehicle types. Deploying jobs to delivery agents, who were selected based on their availability and location and were responsible for on-time delivery. Metrics like delivery time and agent rating are monitored to guarantee client satisfaction. These operations are analysed deeply on the dataset.

1.2 Data Source

The dataset used in this project is sourced from Kaggle, a popular website providing large datasets suited for data analysis and machine learning. The dataset contains data based on agent information, order and delivery details, environmental factors, operational details and customer feedback which are essential to evaluate delivery efficiencies.

1.3 Objectives

The main goal of this project is to develop a model to predict delivery time accurately and to estimate how factors like agent info, locations, timing, weather, traffic, vehicle, area, and delivery type influence delivery timings. The objectives are

- Examining the overall efficiency of the delivery scheduling and comparing them across different circumstances.
- Analysing the performance of the delivery agents based on their ratings, age, and client ratings.
- Identify which environmental factors, weather and traffic conditions are leading to the most delays in deliveries.
- Identification of the impact of vehicle type on delays and efficiencies.
- Explore feature engineering and predictive modelling to anticipate delays.
- Creating visualisation that helps to analyse data easily and is understandable to non-technical viewers
- Building a machine learning model to identify the most influential factors affecting delivery timings.

These help Amazon understand the delays and issues faced by agents, where it optimises operations like better route planning, allocation of suitable vehicles, and scheduling methods to enhance service and customer satisfaction.

1.4 Methodology

The project will follow the below steps to achieve the goal.

- Data understanding and cleaning: The structural analysis of the dataset is done to know the value of the data in the dataset over agents, operations, and environmental features involved. This covers the cleaning process to handle missing values and outlier detection is done based on statistics.
- Data preparation: It will involve scaling the numerical metrics across variables, normalising forms of data, and maintaining the consistency of categorical variables.
- Exploratory Data Analysis (EDA): It includes an in-depth analysis to find the patterns and trends using visualization, correlation heatmaps, and scatterplots between the variables.
- Model development. It divides the data into training and testing to conduct a model evaluation. Utilising Machine Learning, this project used techniques from the Random Forest, Gradient Boosting, and Polynomial Regression to estimate the exact timings of each delivery.

1.5 Significance

Analysing Amazon's delivery operations involves evaluating the delays and obstacles faced by agents in delivering packages to clients, which provides actionable insights to Amazon to make efficient decisions and operational measures to improve the quality of delivery. It can plan dynamic scheduling for agents to make faster delivery while considering weather conditions. It involves providing resource allocation to agents, like suitable vehicles, in terms of extreme geographical factors to agents. Timely delivery of products to customers satisfies customer needs, and this model demonstrates a practical application of predicting real-world logistics, that estimates delivery time across multiple factors accurately.

2 Data Preparation

2.1 Importing Libraries

Importing libraries is essential for data analysis, pre-processing and modelling. **pandas** library used for data manipulation and cleaning, **numpy** used for numerical manipulations. **matplotlib** and **seaborn** for data visualization. **LabelEncoder** for encoding categorical variables into numerical form, **StandardScaler** for scaling variance, **PolynomialFeatures** for generating polynomial. Importing machine learning models like **LinearRegression**, **RandomForestRegressor**, and **GredientBoostingRegressor**. For evaluating the model, **mean_absolute_percentage_error** used ad train_test_split for data splitting.

In [18]:
```
# Import libraries

# Data Cleaning
import pandas as pd
```

```
import numpy as np

# Data Visualisation
import matplotlib.pyplot as plt
import seaborn as sns

# Data preprocessing
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.preprocessing import PolynomialFeatures

# Data Splitting
from sklearn.model_selection import train_test_split

# Model development
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression

# Model Evaluation
from sklearn.metrics import mean_absolute_percentage_error
```

2.2 Reading and Exploring the data

The reading and understanding data is a critical step in data analysis where it involves loading and exploring its structure and gaining overall insights of variables it contains. It sets the foundation for further analysis and helps to identify potentical challenges. Using pandas library, dataset is loaded and pd.read_csv reads the CSV file of dataset inserted.

In [20]:
```
# load the dataset

df = pd.read_csv('amazon_delivery.csv')
```

In [22]:
```
# Overview of dataset

df.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
```

The **info()** function gives an overview of the dataset. There are 43,739 rows and 16 columns. The nine categorical columns have object data type, five columns have float data type and two columns having integer data type.

In [7]:
```
# view the dataset
# the first 5 rows
df.head(5)
```

Out[7]:

	Order_ID	Agent_Age	Agent_Rating	Store_Latitude	Store_Longitude	Drop_Latituc
0	ialx566343618	37	4.9	22.745049	75.892471	22.7650
1	akqg208421122	34	4.5	12.913041	77.683237	13.0430
2	njpu434582536	23	4.4	12.914264	77.678400	12.9242
3	rjto796129700	38	4.7	11.003669	76.976494	11.0536
4	zguw716275638	32	4.6	12.972793	80.249982	13.0127

The **head()** function displays the first five rows in Amazon delivery dataset

In [24]:
```
# the last 5 rows
df.tail()
```

Out[24]:

	Order_ID	Agent_Age	Agent_Rating	Store_Latitude	Store_Longitude	Drop_La
43734	jlxf819993117	30	4.8	26.902328	75.794257	26.9
43735	aevx342135787	21	4.6	0.000000	0.000000	0.0
43736	xnek760674819	30	4.9	13.022394	80.242439	13.0
43737	cynl434665991	20	4.7	11.001753	76.986241	11.0
43738	nsyz997960170	23	4.9	23.351058	85.325731	23.4

The **tail** function displays the last five rows in amazon delivery dataset

In [26]:
```
# summarises the dataset

df.describe()
```

Out[26]:

	Agent_Age	Agent_Rating	Store_Latitude	Store_Longitude	Drop_Latitude	Drop_L
count	43739.000000	43685.000000	43739.000000	43739.000000	43739.000000	437
mean	29.567137	4.633780	17.210960	70.661177	17.459031	
std	5.815155	0.334716	7.764225	21.475005	7.342950	
min	15.000000	1.000000	-30.902872	-88.366217	0.010000	
25%	25.000000	4.500000	12.933298	73.170283	12.985996	
50%	30.000000	4.700000	18.551440	75.898497	18.633626	
75%	35.000000	4.900000	22.732225	78.045359	22.785049	
max	50.000000	6.000000	30.914057	88.433452	31.054057	

The **describe()** function gives a summary of the dataset. Agent age ranges from minimum of 15 years to maximum of 50 years. The mean age is 29.5 which is 30 years. 25% of agents are younger than 25 years and 75% of agents are younger than 35 years. The standard deviation of agent age is 5.81. The agent rating ranges from minimum of 1 and maximum of 6 where 4.6 is average ratings. The standard deviation is 0.33 which implies most agents perform consistently well. 25% of ratings are 4.5 and 75% are 4.9 ratings. Store Latitude ranges from -30.90 to 30.91 with an average of 17.21 and Store longitude ranges from -88.37 to 88.43 with average of 70.66 . The standard deviation of latitude, 7.7 and longitude 21.47 represents diversity in store locations. Drop latitude ranges from 0.01 to 31.05 with average of 17.45 and drop longitude ranges from 0.01 to 88.56 with average of 70.8. The standard deviation of drop locations are same as store locations. Delivery time ranges from minimun of 10 minutes to maximun of 270 minutes with average of 125 minutes. 25% of time is 90 minutes and 75% of time is 160 minutes where standard deviation is 51.9. These key insights help to find outliers in dataset and identify patterns for investigation.

3. Data Cleaning and Manipulation

Data cleaning is the process of handling the missing values and removing irrelevant data from the dataset. This is to ensure data quality and reliability for data analysis.

In [14]:
```
# view the missing values in the dataset
def check_missing_values(df):
    missing_values = df.isnull().sum()
    if missing_values.sum() == 0:
        print("No missing values found.")
    else:
        print("Missing values:")
        print(missing_values[missing_values > 0])

check_missing_values(df)
```

```
Missing values:
Agent_Rating    54
Weather         91
dtype: int64
```

The function **check_missing_values** checks the missing values in the DataFrame df. It calculates total number of missing values in each column of the dataset. In this dataset, it has some missing values so if condition is false, and else block get executed. Agent rating column has 54 missing values and weather column has 91 missing values.

In [15]:
```
# check for duplicate rows
def check_duplicates(df):
    duplicates = df.duplicated().sum()
    if duplicates > 0:
        print("Duplicates found in the dataset:", duplicates)
    else:
```

```
        print("No duplicates found in the dataset.")

check_duplicates(df)
```

```
No duplicates found in the dataset.
```

The **check_duplicates** function is used to check the duplicate rows in dataset. It calculates the total number of duplicate rows. In this dataset, it does not have any duplicate rows so if condition is false and else block is executed.
Removing duplicates in dataset reduces data size and improves quality and performance where it ensures unique records for analysis.

In [13]:
```
# Proceed to developing the classes for cleaning the data
# There are missing values in the dataset

# Begin by identifying the columns which are numeric and categorical and then fill

def identify_column_types(df):
    numerical_columns = df.select_dtypes(include=['int64', 'float64']).columns
    categorical_columns = df.select_dtypes(include=['object']).columns
    return numerical_columns,categorical_columns

# Access the function to identify the columns
numerical_columns, categorical_columns = identify_column_types(df)

# print the numerical columns
print("\nNumerical columns:")
for col in numerical_columns:
    print(col)

# print the categorical columns
print("\nCategorical columns:")
for col in categorical_columns:
    print(col)
```

```
Numerical columns:
Agent_Age
Agent_Rating
Store_Latitude
Store_Longitude
Drop_Latitude
Drop_Longitude
Delivery_Time

Categorical columns:
Order_ID
Order_Date
Order_Time
Pickup_Time
Weather
Traffic
Vehicle
Area
Category
```

The **identify_column_types** function is used to classify the data frame columns into numerical and categorical columns based on their data types. First, it filters the dataframe to include columns which have int64 or float64 data types and then retrieves the columns and classify as numerical. Next, it filters the df to include columns having only object data types and which retrieve and classify as categorical. The functions **numerical_columns** and **categorical_column**, column name for numerical data and categorical data are used to assess the function. Then it returns the columns based on their data types.
This function is used for data preprocessing, feature engineering and model preparation where it help to identify quickly and handled properly for machine learning models.

3.1 Handling missing values

Handling missing values is a vital step in data preprocessing of data analysis. If it is not handled properly, it may lead to biased results, reduced accuracy and unreliable insights

In [ ]:
```
# fill null values with numerical columns with mean and categorical columns with mo

def fill_null_values(df):
    for col in numerical_columns:
        if df[col].isnull().any():
            fill_value = df[col].mean()
            df[col] = df[col].fillna(fill_value)
    for col in categorical_columns:
        if df[col].isnull().any():
            fill_value = df[col].mode()[0]
            df[col] = df[col].fillna(fill_value)

    print(f"\nFilling of null values complete!")

    # Re-check for null values
    print(f"\nChecking for null values:")
    null_values =df.isnull().sum()
    if null_values.sum() == 0:
        print("No null values found")
    else:
        print(null_values[null_values > 0])


df = fill_null_values(df)
```

```
Filling of null values complete!

Checking for null values:
No null values found
```

The **fill_null_values** function is used to handle missing values in the dataframe (df) which fills appropriate values. Already, we have classified all columns into numerical and categorical. First, it checks null values in each column of numerical column, if column has missing values, it calculates the mean of the column using **df[col].mean()** and replaces the missing values with mean using **df[col].fillna(fill_value)**. Next for catgeorical columns, it checks the null

values for each column. If the column has a missing value, it calculates the most frequent value that is mode of the particular column using **df[col].mode()[0]** and replaces the missing values with mode using **df[col].fillna(fill_value)**. Then it displays the completion of filling null values. Again it checks for null values and return no null values in the dataset.
This step is essential for data preparation for analysis and maintains consistency of data types. It ensures data completeness by filling with statistically meaningful values and more reliable for further analysis and machine learning tasks.

In [20]:
```
# Check datatypes in each column to determine more cleaning needed
# There could be columns with mixed datatypes

df.dtypes
```

Out[20]:
```
Order_ID            object
Agent_Age            int64
Agent_Rating       float64
Store_Latitude     float64
Store_Longitude    float64
Drop_Latitude      float64
Drop_Longitude     float64
Order_Date          object
Order_Time          object
Pickup_Time         object
Weather             object
Traffic             object
Vehicle             object
Area                object
Delivery_Time        int64
Category            object
dtype: object
```

The **dtypes** function is used to display the data types of all columns in dataframe (df). Columns agent_Age and Delivery_time have integer data type where it represents age and delivery time in minutes in numerical values. Columns like Agent_Rating, Store_Latitude, Store_Longitude, Drop_Latitude, Drop_Longitude have float data types suitable for continuous values requiring decimal points. Columns like Order_ID, Order_Data, Pickup_Time, Weather, Traffic, Vehicle, Area and Category have object that is string data type which include text and other non-numerical data.
This function is used to identify potential issues like mixed data types in a column which can complicate in model training. It helps to preprocess data and validate expected data types and also gives a clear understanding of dataset structure.

In [77]:
```
# check unique values in the dataset

for col in df.columns:
    print(f"\nUnique values in {col}: {df[col].unique()}")
```

```
Unique values in Order_ID: ['ialx566343618' 'akqg208421122' 'njpu434582536' ... 'xne
k760674819'
 'cynl434665991' 'nsyz997960170']

Unique values in Agent_Age: [37 34 23 38 32 22 33 35 36 21 24 29 25 31 27 26 20 28 3
9 30 15 50]

Unique values in Agent_Rating: [4.9 4.5 4.4 4.7 4.6 4.8 4.2 4.3 4.  4.1 5.  3.5 3.8
nan 3.9 3.7 2.6 2.5
 3.6 3.1 2.7 1.  3.2 3.3 6.  3.4 2.8 2.9 3. ]

Unique values in Store_Latitude: [ 22.745049  12.913041  12.914264  11.003669  12.97
2793  17.431668
  23.369746  12.352058  17.433809  30.327968  10.003064  18.56245
  30.899584  26.463504  19.176269  12.311072  18.592718  17.426228
  22.552672  18.563934  23.357804  12.986047  19.221315  13.005801
  26.849596  21.160522  12.934179  18.51421   11.022477  21.160437
  15.51315   15.561295   0.        18.55144   18.593481  21.173343
  17.451976  12.972532  13.064181  21.149569  19.091458  22.539129
  12.970324  21.175975  11.003681  10.96185   27.165108  26.88842
  26.913987  12.3085    21.183434  19.254567  25.449659  30.372202
  21.157735  21.186438  17.431477  12.933298  22.311358  12.934365
  13.086438  26.913483  17.411028  18.516216  15.5696    19.876428
  12.939496  23.374878 -27.163303  22.74806   26.891191  12.316967
  18.927584  11.022298  12.325461  18.530963  17.458998  19.1813
  22.727021  12.979166  18.994237  12.284747  11.000762  26.479108
  21.175104  12.975377  23.359194  12.297954  23.359407  23.416792
  26.892312  22.761593  11.026117  26.483042  23.359033  22.569358
  19.121999  21.157729  22.311603  18.536562  22.745536  17.41233
  21.17106   21.186884  19.065838  12.98041   12.310972  11.025083
  22.538731  11.001753  22.310237  22.725748  17.450851  13.081878
  10.035573  22.308096  12.975996  19.178321  26.910262  25.443994
  12.321214  19.055831  12.337928  23.351489  22.728163  13.026286
  26.766536  21.149834  21.149669  26.482581  26.911378  13.02978
  12.323194  22.569367  12.949934  22.725835  30.89286   12.933284
  17.429585  12.978453  13.044694  26.846156  26.911927  10.020683
  12.935662  15.585658  22.310526  19.103249  26.471617  19.866969
  12.906229  22.751857  11.010375  17.483216  27.160934  11.006686
  25.457687  19.12663   19.875016  13.026279  17.455894  12.979096
  27.163303  26.482419  22.307898  27.160832  10.027014  30.890184
  21.15276   22.753839  19.876106  30.328174  13.058616 -27.165108
  21.170096  30.905562  26.902328  26.90519   17.428294  12.323978
  26.492106  19.003517  10.994136  17.440827  18.543626  12.304569
  12.299524  22.31279   19.131141  19.879631  18.520016  21.170798
  22.760072  22.761226  23.354422  22.722634  22.526461  23.371292
  13.029198  18.546258  26.921411  11.008638  12.972161  13.049645
  17.430448  19.874733  13.045479  26.956431  13.022394  22.75004
  21.173493  13.091809  18.533811  15.496162  26.913726  30.895817
  19.120083  27.161661  13.066762  30.362686  17.410371   9.979186
  18.539299  22.310329  11.022169  19.875908  10.006881  12.326356
  19.875522  22.552996  30.895204  25.450329  18.546947  30.340722
  22.651847  25.45235   17.424114  12.323225  26.905287  17.422819
  17.45971   22.695207  18.569156  11.003008  26.914142  30.359722
  12.334022  23.214294  12.970221  22.753659  18.636215  23.234249
  25.457116  22.744648  23.39925   19.874103  22.577821  12.337978
  22.514119  22.311844  18.994049  22.732225  30.914057  23.233219
```

```
  12.981615  23.184992  26.49095   22.551084  11.021278  30.899992
  11.024839  23.351058  19.876219  19.880256  30.893234  30.319528
  27.195928  25.450317  21.185047  23.355164  23.232537  18.53408
  26.902908  11.016298  10.000706  25.454697  21.186608  26.90294
  23.353783  23.214459  23.374989  30.893244  30.885915  17.438263
  15.544419  22.751234  26.474986   9.970717  30.873988  19.888716
  13.054347  22.547186  18.536718  23.232357  19.1093    22.32
  26.469003  26.483672  10.027364 -15.546594  19.207222  19.874449
  12.323994  11.02091   19.22384   30.366322  30.361281  26.473698
  18.554382  22.553227  30.332735   9.982834  15.498603  22.5491
  25.450377  26.471529  23.218998  23.235123  22.533662  30.885814
  15.574828  25.459775  30.893081  22.514688  23.266261  23.333017
  27.157772  15.516833  13.027018  22.527893  15.546594  15.157944
  27.16185   30.88722   27.158822 -23.230791  23.264015  27.161694
   9.991703  27.201725   9.960846  23.234631  26.481547 -22.539129
  15.303897   9.988483  25.454648  30.342509 -15.157944  22.53796
  -9.959778  30.346994 -15.51315   15.49395   11.001852  30.893384
 -15.576683  19.875337  15.556561  23.230791 -26.891191  15.56155
  26.47775  -30.902872  30.335259   9.985497  25.451517  26.472001
   9.959778  -9.988483 -23.211529  30.892978  15.506205  19.878028
 -30.895817 -26.474986  22.538999   9.957144  22.515082  25.451646
 -15.5696   -12.970324   9.979363  10.028047  26.474133  30.902872
   9.985697  23.211529  19.876994  27.159795  15.576683  22.514585
  25.449872  25.453436   9.966783 -17.451976 -26.472001 -26.910262
 -25.449872 -27.159795 -22.553227  -9.966783 -19.878028  -9.970717
 -22.526461 -19.876994 -25.449659 -12.352058 -25.450317 -26.49095
 -30.319528 -10.028047 -19.876106  26.47     -26.463504 -30.899584
 -23.234631 -23.234249 -15.496162 -23.374989 -22.538999 -30.885915
 -22.53796  -23.184992 -10.035573 -23.235123 -25.453436 -23.214459
 -26.483042 -21.183434 -19.879631 -19.874449 -10.020683 -19.875337
 -23.416792 -19.875522 -19.875016 -22.569358 -27.201725 -19.221315
 -22.538731 -27.160832 -13.066762 -30.885814 -25.451517 -22.651847
 -22.547186 -26.474133 -30.327968 -26.481547 -22.515082 -23.351058
 -26.473698 -30.873988 -22.514119 -19.876428 -13.02978  -21.170096
  -9.960846 -15.303897 -27.16185  -30.328174 -22.5491   -26.471529
 -10.994136 -30.893244 -19.888716 -22.551084 -12.935662 -15.561295
  -9.979186 -12.975377 -30.346994 -25.454697 -30.372202 -19.880256
 -19.103249 -10.027014 -30.361281 -10.003064 -22.552996  -9.982834
 -19.866969 -26.469003 -22.577821 -22.569367 -23.232357 -13.091809
 -22.533662 -27.158822 -19.875908 -26.479108 -30.892978 -23.232537
 -22.514585 -17.426228 -18.592718 -15.56155  -19.091458 -10.96185
 -25.457687 -15.506205 -15.516833 -12.3085   -13.049645 -26.47
 -25.443994 -12.939496 -26.482581 -15.498603 -19.874733]

Unique values in Store_Longitude: [ 75.892471  77.683237  77.6784    76.976494  80.2
49982  78.408321
  85.33982   76.60665   78.386744  78.046106  76.307589  73.916619
  75.809346  80.372929  72.836721  76.654878  73.773572  78.407495
  88.352885  73.915367  85.325146  80.218114  72.862381  80.250744
  75.800512  72.771477  77.615797  73.838429  76.995667  72.774209
  73.78346   73.749478   0.        73.804855  73.785901  72.792731
  78.385883  77.608179  80.236442  72.772697  72.827808  88.365507
  77.645748  72.795503  76.975525  76.971082  78.015053  75.800689
  75.752891  76.665808  72.814492  72.848923  81.839744  78.077151
  72.768778  72.794115  78.40035   77.614293  73.164798  77.616155
  80.220672  75.803139  78.329645  73.842527  73.742294  75.364792
```

```
 77.625999  85.335739  78.057044  75.8934    75.802083  76.603067
 72.832585  76.998349  76.632278  73.828972  78.500366  72.836191
 75.884167  77.640709  72.825553  76.625861  76.981876  80.315042
 72.804342  77.696664  85.325447  76.665169  85.325055  85.316842
 75.806896  75.886362  76.944652  80.317833  85.325347  88.433452
 72.908493  72.768726  73.165012  73.896485  75.893106  78.449654
 72.789292  72.793616  72.832658  77.640489  76.659264  77.015393
 88.364878  76.986241  73.158921  75.898497  78.379347  80.248519
 76.336958  73.167753  80.221898  72.834715  75.783013  81.860187
 76.621094  72.833984  76.617889  85.324253  75.884212  80.275235
 75.837333  72.778666  72.772629  80.315628  75.789034  80.208812
 76.630583  88.433187  77.699386  75.887648  75.822199  77.615428
 78.392621  77.643685  80.26147   75.8023    75.797282  76.310631
 77.61413   73.743606  73.170937  72.846749  80.313564  75.318894
 77.596791  75.866699  76.95295   78.552111  78.044095  76.951736
 81.835585  72.829976  75.322405  80.174568  78.375467  77.640625
 80.320939  73.167788  78.011608  76.308053  75.829615  72.778059
 75.897429  75.340775  78.049117  80.264151  72.789122  75.832841
 75.794257  75.810753  78.404423  76.627961  80.327797  72.82765
 76.963303  78.393391  73.905101  76.643622  76.64262   73.170283
 72.813074  75.323403  73.830547  72.790489  75.892574  75.887522
 85.3329    75.886959  88.364453  85.327872  77.570997  73.904337
 75.793604  76.984311  77.596014  80.242268  78.418213  75.353942
 80.23311   75.776649  80.242439  75.902847  72.801953  80.219104
 73.899315  73.825364  75.75282   75.813112  72.907385  78.011544
 80.251865  78.06889   78.437225  76.317361  73.897902  73.169083
 76.999594  75.358888  76.345397  76.619103  75.367127  88.35231
 75.822103  81.834279  73.900626  78.060221  75.881991  81.841889
 78.347554  76.630028  75.794592  78.449578  78.368855  75.866059
 73.774723  76.97544   75.805704  78.067079  76.618203  77.435361
 77.645396  75.903365  73.751081  77.434007  81.859682  75.894377
 85.390464  75.368419  88.400581  76.616792  88.362504  73.165081
 72.825203  75.874765  75.83982   77.433571  80.231598  77.417227
 80.318656  88.354127  76.995017  75.831338  77.007003  85.325731
 75.346017  75.323503  75.82172   78.040267  77.998092  81.831681
 72.80859   85.324097  77.429845  73.89852   75.792934  76.972076
 76.349516  81.834492  72.794136  75.793007  85.326967  77.434976
 85.335486  75.821817  75.788259  78.397865  73.755736  75.88949
 80.342796  76.285447  75.842739  75.321461  80.257221  88.35068
 73.830327  77.429989  72.825451  73.17      80.316344  80.320708
 76.308258  73.760431  72.972281  75.360232  76.626167  76.940432
 72.841347  78.070453  78.068022  80.352677  73.798206  88.353273
 78.054222  76.283268  73.826911  88.400467  81.834236  80.313458
 77.373573  77.398886  88.366217  75.786976  73.766883  81.834841
 75.821495  88.393294  77.379605  85.3172    78.04725   73.768172
 80.254791  88.368628  73.950889  78.040165  75.804893  78.045359
 77.43702   77.408236  78.034714  76.293136  78.007553  76.293936
 77.401663  80.299775  73.914336  76.295211  81.834502  78.061187
 88.349843  76.296106  78.062543  73.827423  76.976268  75.821202
 73.75575   75.316722  73.763633  73.749092  80.351569  75.826808
 78.053162  76.276999  81.832616  80.354002  77.419399  75.821847
 73.766668  75.317475  88.322337  76.296783  88.36783   81.832796
-77.645748  76.285001  76.310019  80.3481    76.281128  75.372353
 78.04299   88.39331   81.836167  81.833167  76.242981 -78.385883
-75.783013 -78.04299  -76.60665   80.35     -85.335486 -72.814492
-85.316842 -80.251865 -75.881991 -85.325731 -80.208812 -72.789122
```

```
 -76.963303 -77.61413  -77.696664 -72.846749 -80.316344 -88.366217
 -78.045359 -78.407495 -73.773572 -76.971082 -76.665808 -80.242268
 -80.318656]

Unique values in Drop_Latitude: [22.765049 13.043041 12.924264 ... 30.940184 30.9828
72 15.636205]

Unique values in Drop_Longitude: [75.912471 77.813237 77.6884   ... 75.879615 75.906
808 73.896668]

Unique values in Order_Date: ['2022-03-19' '2022-03-25' '2022-04-05' '2022-03-26' '2
022-03-11'
 '2022-03-04' '2022-03-14' '2022-03-20' '2022-02-12' '2022-02-13'
 '2022-02-14' '2022-04-02' '2022-03-01' '2022-03-16' '2022-02-15'
 '2022-03-10' '2022-03-27' '2022-03-12' '2022-04-01' '2022-03-05'
 '2022-02-11' '2022-03-08' '2022-04-03' '2022-03-30' '2022-03-28'
 '2022-03-18' '2022-04-04' '2022-03-24' '2022-03-09' '2022-03-02'
 '2022-03-13' '2022-03-29' '2022-03-31' '2022-03-17' '2022-03-07'
 '2022-03-15' '2022-02-16' '2022-03-03' '2022-02-18' '2022-03-23'
 '2022-02-17' '2022-03-06' '2022-03-21' '2022-04-06']

Unique values in Order_Time: ['11:30:00' '19:45:00' '08:30:00' '18:00:00' '13:30:00'
'21:20:00'
 '19:15:00' '17:25:00' '20:55:00' '21:55:00' '14:55:00' '17:30:00'
 '09:20:00' '19:50:00' '20:25:00' '20:30:00' '20:40:00' '21:15:00'
 '20:20:00' '22:30:00' '08:15:00' '19:30:00' '12:25:00' '18:35:00'
 '20:35:00' '23:20:00' '23:35:00' '22:35:00' '23:25:00' '13:35:00'
 '21:35:00' '18:55:00' '14:15:00' '11:00:00' '09:45:00' '08:40:00'
 '23:00:00' '19:10:00' '10:55:00' '21:40:00' '19:00:00' '16:45:00'
 '15:10:00' '22:45:00' '22:10:00' '20:45:00' '22:50:00' '17:55:00'
 '09:25:00' '20:15:00' '22:25:00' '22:40:00' '23:50:00' '15:25:00'
 '10:20:00' '10:40:00' '15:55:00' '20:10:00' '12:10:00' '15:30:00'
 '10:35:00' '21:10:00' '20:50:00' '12:35:00' '21:00:00' '23:40:00'
 '18:15:00' '18:20:00' '11:45:00' '12:45:00' '23:30:00' '10:50:00'
 '21:25:00' '10:10:00' '17:50:00' '22:20:00' '12:40:00' '23:55:00'
 '10:25:00' '08:45:00' '23:45:00' '19:55:00' '22:15:00' '23:10:00'
 '09:15:00' '18:25:00' '18:45:00' '16:50:00' '00:00:00' '14:20:00'
 '10:15:00' '08:50:00' '09:00:00' '17:45:00' '16:35:00' '21:45:00'
 '19:40:00' '14:50:00' '18:10:00' '12:20:00' '12:50:00' '09:10:00'
 '12:30:00' '17:10:00' '17:20:00' '18:30:00' '13:10:00' '19:35:00'
 '09:50:00' '15:00:00' '20:00:00' '10:30:00' '09:40:00' '15:35:00'
 '16:55:00' '22:55:00' '16:00:00' '17:15:00' '21:30:00' '18:40:00'
 '11:10:00' '13:50:00' '10:00:00' '21:50:00' '11:50:00' '22:00:00'
 '08:25:00' '11:20:00' '11:55:00' '09:30:00' '08:20:00' '08:10:00'
 '11:40:00' '23:15:00' '19:20:00' '12:15:00' '11:35:00' '11:15:00'
 '17:35:00' '17:40:00' '14:40:00' '18:50:00' '11:25:00' '14:25:00'
 '12:00:00' '16:10:00' '19:25:00' '08:55:00' '13:40:00' '17:00:00'
 '09:35:00' '08:35:00' '16:15:00' '13:20:00' '15:50:00' '15:20:00'
 '16:20:00' '14:30:00' '15:45:00' '16:40:00' '13:00:00' '12:55:00'
 '10:45:00' '13:25:00' '09:55:00' '15:15:00' '13:15:00' '14:00:00'
 '15:40:00' '16:25:00' '14:10:00' '13:45:00' '13:55:00' '14:35:00' 'NaN '
 '16:30:00' '14:45:00']

Unique values in Pickup_Time: ['11:45:00' '19:50:00' '08:45:00' '18:10:00' '13:45:0
0' '21:30:00'
 '19:30:00' '17:30:00' '21:05:00' '22:10:00' '15:05:00' '17:40:00'
```

```
 '09:30:00' '20:05:00' '20:35:00' '15:10:00' '20:40:00' '20:50:00'
 '20:25:00' '22:45:00' '08:30:00' '19:45:00' '12:30:00' '18:50:00'
 '23:30:00' '21:35:00' '23:45:00' '22:50:00' '22:40:00' '23:35:00'
 '13:40:00' '21:45:00' '19:10:00' '14:25:00' '11:10:00' '09:55:00'
 '08:55:00' '23:10:00' '19:25:00' '11:00:00' '19:15:00' '16:55:00'
 '11:40:00' '15:15:00' '22:55:00' '22:25:00' '20:55:00' '23:05:00'
 '18:00:00' '23:00:00' '09:40:00' '20:20:00' '22:35:00' '22:00:00'
 '23:55:00' '15:40:00' '10:30:00' '21:00:00' '10:50:00' '16:05:00'
 '20:15:00' '12:15:00' '15:45:00' '22:15:00' '10:45:00' '00:05:00'
 '21:25:00' '12:45:00' '21:15:00' '18:20:00' '18:25:00' '11:50:00'
 '12:50:00' '10:55:00' '21:40:00' '10:20:00' '17:55:00' '23:50:00'
 '12:55:00' '00:10:00' '10:40:00' '09:00:00' '20:45:00' '20:00:00'
 '23:15:00' '18:35:00' '22:20:00' '17:00:00' '00:15:00' '21:20:00'
 '14:35:00' '10:25:00' '09:05:00' '16:50:00' '08:40:00' '23:40:00'
 '21:50:00' '19:55:00' '15:00:00' '10:35:00' '09:25:00' '17:20:00'
 '17:25:00' '20:10:00' '00:00:00' '17:35:00' '19:00:00' '19:05:00'
 '13:20:00' '18:05:00' '19:20:00' '10:05:00' '09:10:00' '21:55:00'
 '19:40:00' '09:50:00' '15:50:00' '18:30:00' '18:15:00' '16:15:00'
 '11:15:00' '21:10:00' '15:30:00' '22:30:00' '15:20:00' '23:20:00'
 '11:25:00' '13:55:00' '18:45:00' '22:05:00' '11:55:00' '18:55:00'
 '18:40:00' '09:45:00' '17:15:00' '12:05:00' '12:00:00' '19:35:00'
 '08:25:00' '11:05:00' '15:35:00' '12:40:00' '12:25:00' '08:20:00'
 '23:25:00' '16:10:00' '17:50:00' '08:15:00' '20:30:00' '11:20:00'
 '08:50:00' '14:45:00' '17:45:00' '10:00:00' '08:35:00' '11:35:00'
 '14:30:00' '12:10:00' '10:15:00' '17:05:00' '10:10:00' '09:35:00'
 '11:30:00' '16:25:00' '09:15:00' '13:35:00' '15:55:00' '13:00:00'
 '13:10:00' '13:05:00' '16:20:00' '16:30:00' '16:45:00' '09:20:00'
 '13:25:00' '14:15:00' '16:35:00' '16:40:00' '14:05:00' '13:30:00'
 '14:40:00' '12:20:00' '13:50:00' '17:10:00' '14:20:00' '12:35:00'
 '14:00:00' '14:10:00' '15:25:00' '14:55:00' '13:15:00' '14:50:00'
 '16:00:00']

Unique values in Weather: ['Sunny' 'Stormy' 'Sandstorms' 'Cloudy' 'Fog' 'Windy' nan]

Unique values in Traffic: ['High ' 'Jam ' 'Low ' 'Medium ' 'NaN ']

Unique values in Vehicle: ['motorcycle ' 'scooter ' 'van' 'bicycle ']

Unique values in Area: ['Urban ' 'Metropolitian ' 'Semi-Urban ' 'Other']

Unique values in Delivery_Time: [120 165 130 105 150 200 160 170 230 115 100 205  33
75 180 195  90 190
 235  60 110 125  35  50  95  55 140 260  16  80 135 245  85  70  15 185
 220 210 155  65 145  19 175 215 240  26 270  25  53  28 250  24  37  13
 225  10  22  21  14  12  27  29  23  41  34  42  11  45  47 265 255  17
  38  18  36  20  39  44  49  48  46  32  40  43  30  51  31  52  54]

Unique values in Category: ['Clothing' 'Electronics' 'Sports' 'Cosmetics' 'Toys' 'Sn
acks' 'Shoes'
 'Apparel' 'Jewelry' 'Outdoors' 'Grocery' 'Books' 'Kitchen' 'Home'
 'Pet Supplies' 'Skincare']
```

The **unique()** function gives unique values present in each column. For example, Weather column has sunny, stormy, sandstorms, cloudy, fog, and windy conditions. Traffic column has high, jam, low and medium conditions. Vehicle column has motorcycle, scooter, van and

bicycle types. Category column has various characteristics of products purchased by the customers.
This function is used to understand the data which provides diversity of values in each column. It helps to identify categorical variables and anomalies.

Looking at the unique values in each columns there are "NaN" values in the columns "Order_time" and "Traffic" columns. We will replace them with the mode respectively.

In [ ]:
```
# Function to replace NaN values with the mode of the column
def replace_nan_with_mode(df):
    for col in df.columns:

        # Check for both string 'NaN' values and actual NaN values as some have spa
        check_condition = df[col].astype(str).str.strip() == 'NaN'

        # Replace both 'NaN' strings and np.nan with the mode value
        if check_condition.any():
            mode_value = df[check_condition == False][col].mode()[0]

            # Replacing both 'NaN' and 'NaN ' with the mode value
            df[col] = df[col].replace({'NaN': mode_value, 'NaN ': mode_value})

        # Recheck for any remaining NaN values
    if check_condition.any():
        print(f"Warning: There are still NaN values in column '{col}'.")
    else:
        print(f"No NaN values remaining in column")


replace_nan_with_mode(df)
```

```
Success: No NaN values remaining in column
```

The **replace_nan_with_mode** function is used to handle missing values in dataframe by using mode. Each column is iterated to check missing values and **df[col].astype(str).str.strip() == 'NaN'** identifies rows in columns where the value 'NaN' is used. Then, the mode value is calculated regarding the columns using **df[col].mode()**. Then 'NaN are replaced by **.replace()** with computed mode.
Rechecking for remaining NaN values where it confirms no NuN values in the dataset.
Thius ensures that columns are cleaned for further analysis.

Data cleaning completed. As there are no null values,no duplicate values and no unusual data elements in the data, we can proceed to the next step.

4. Exploratory Data Analysis

Exploratory Data Analysis(EDA) is an essential step which focuses on summarising the main characteristics of the dataset, identifying patterns and uncovering relationships among variables. It serves as a bridge between raw data and actionable insights, guiding the

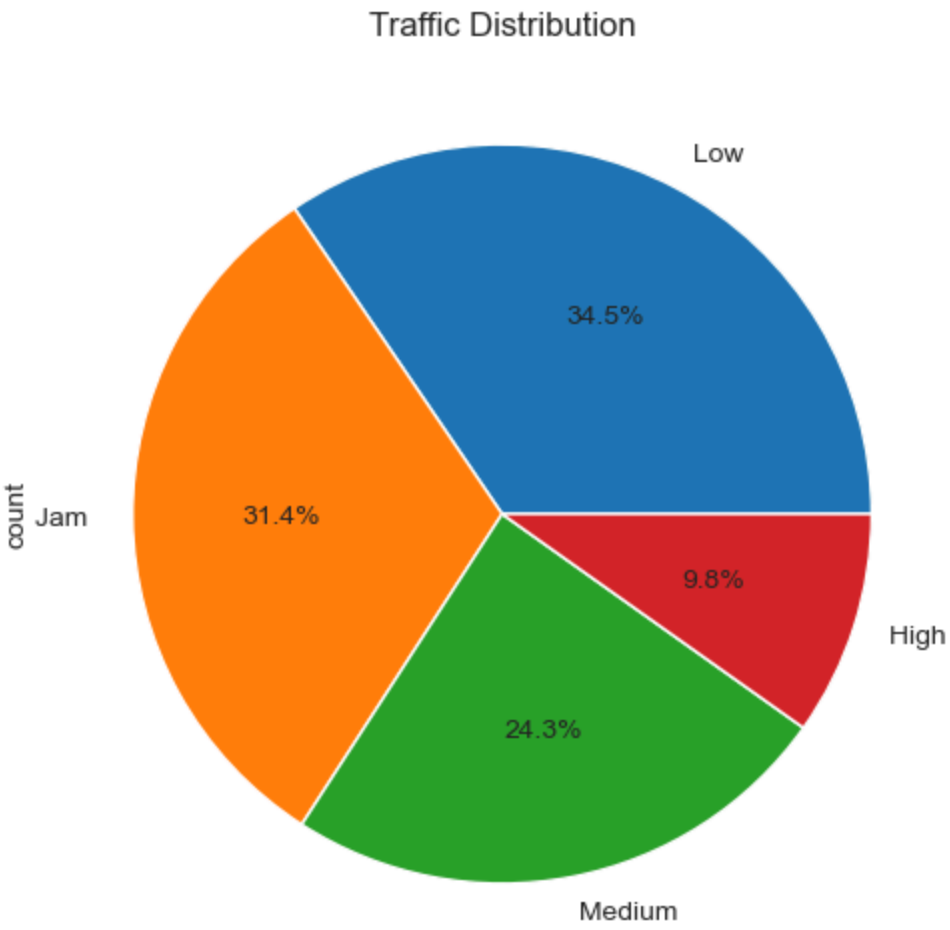
decision-making process and model-building phases.
It enhances the prediction power and ensures selecting relevant features for modeling.It helps to validate assumptions about the data.

In [34]:
```
# Import libraries
import matplotlib.pyplot as plt
import seaborn as sns
```

In [111…
```
# Setting default structure for plotting the graphs

# Set white background for plots to make them more readable
sns.set_style("whitegrid")

# Set the size of the plots, rcParams is a dictionary of matplotlib parameters
# sets a default figure size for the plots
plt.rcParams['figure.figsize'] = (10, 6)

# Generic function for pie charts
def plot_pie(column, title):
    plt.figure()
    df[column].value_counts().plot(kind='pie', autopct='%1.1f%%')
    plt.title(title)
    plt.show()

# Generic function for bar charts
def plot_distribution(column, title):
    plt.figure()
    sns.countplot(data=df, x=column)
    plt.title(title)
    plt.xticks(rotation=45)
    plt.show()

# Generic function for multiple elements for a bar plot
def plot_multiple_elements(column_1,column_2, title):
    plt.figure()
    sns.barplot(data=df, x=column_1, y=column_2)
    plt.title(title)
    plt.xticks(rotation=45)
    plt.show()

# Generic function for box plots
def plot_boxplot(column_s1,column_s2, title):
    plt.figure()
    sns.boxplot(data=df, x=column_s1, y=column_s2)
    plt.title(title)
    plt.xticks(rotation=45)
    plt.show()
```

In [74]:
```
# Plot for Traffic
def plot_traffic(df):
    plot_pie('Traffic', 'Traffic Distribution')
```

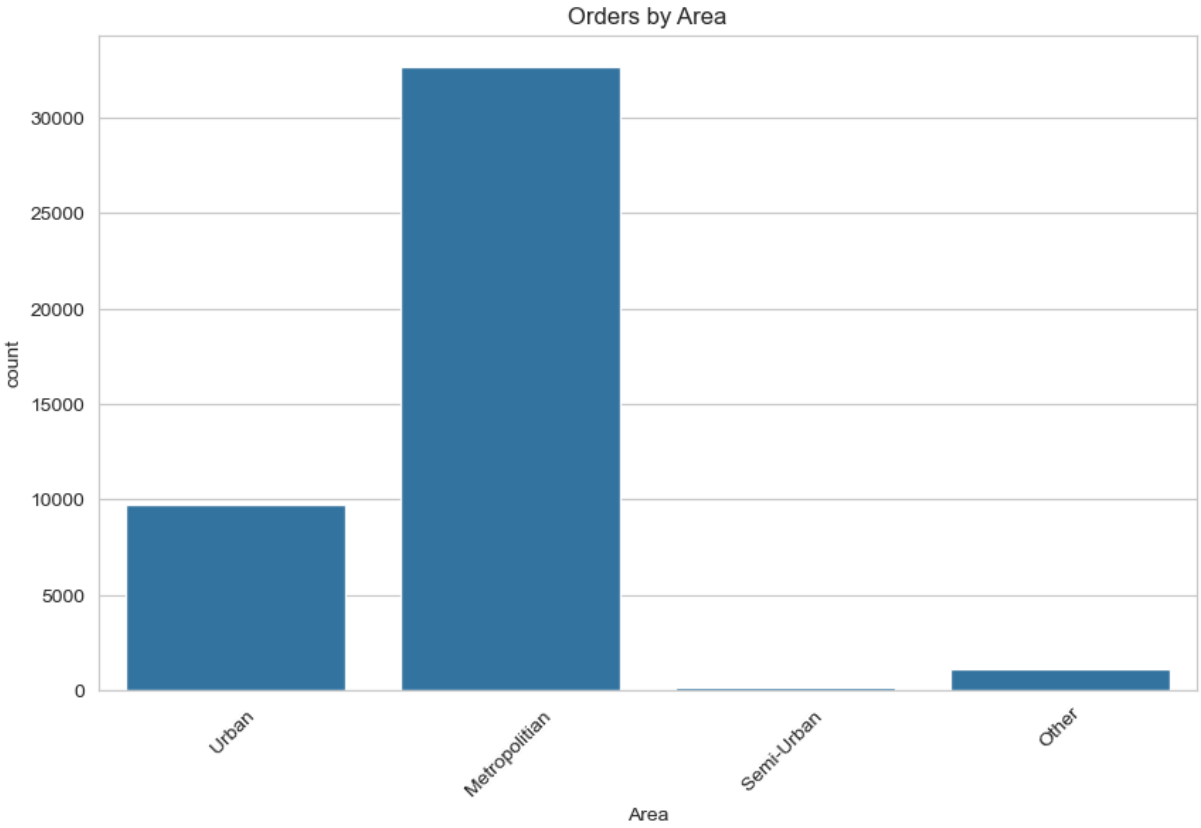
```
plot_traffic(df)
```



The pie chart depicts the distribution of traffic levels across four categories: Low, Medium, High, and Jam. The "Low" category constitutes the largest portion at 34.5%, followed by "Jam" at 31.4%, and "Medium" at 24.3%, while "High" traffic accounts for the smallest share at 9.8%. This distribution indicates that a significant proportion of traffic conditions are either at low levels or in traffic jams, suggesting a potential disparity in traffic flow efficiency. The relatively low percentage of "High" traffic may indicate that most areas experience either smooth traffic or severe congestion rather than moderate delays. This pattern could reflect structural inefficiencies in the road network or peak-time travel patterns. Understanding these distributions is crucial for designing targeted interventions to reduce congestion and improve overall traffic flow.

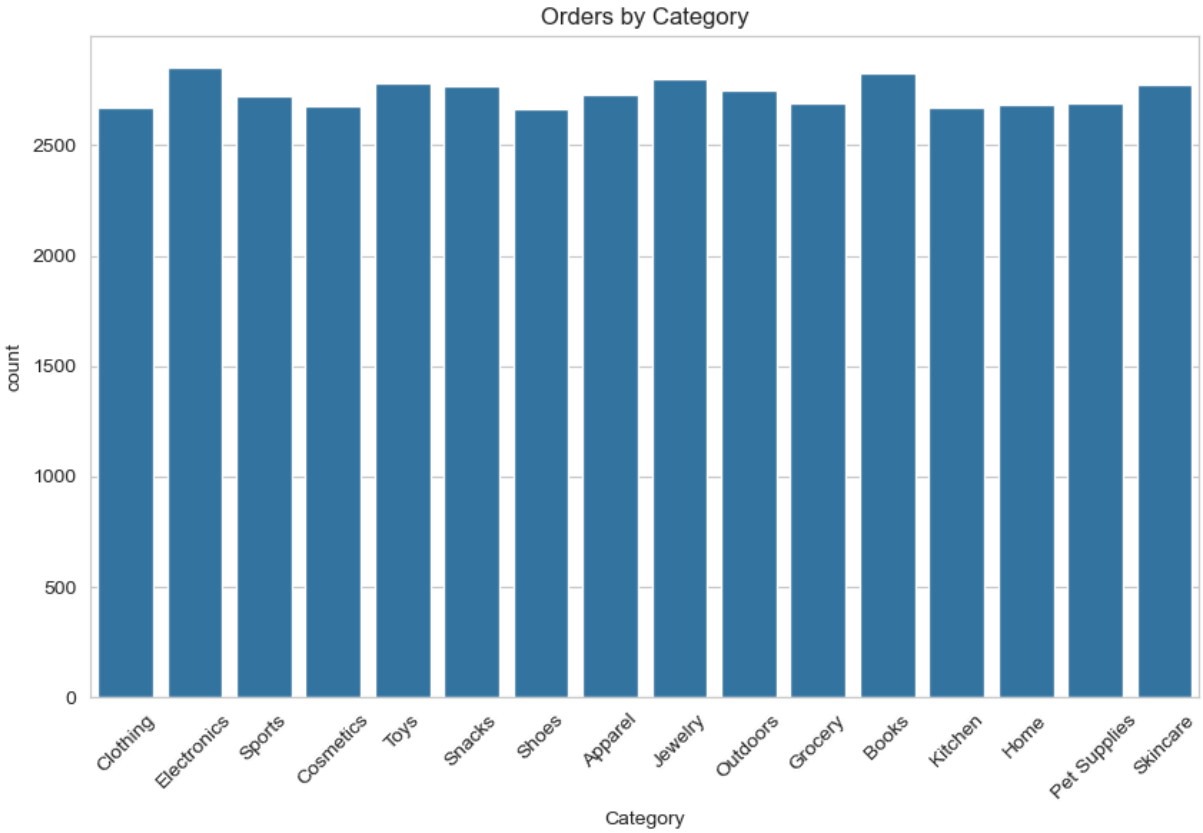
In [119…

```
# plot for Area
def plot_area(df):
    plot_distribution('Area', 'Orders by Area')

plot_area(df)
```



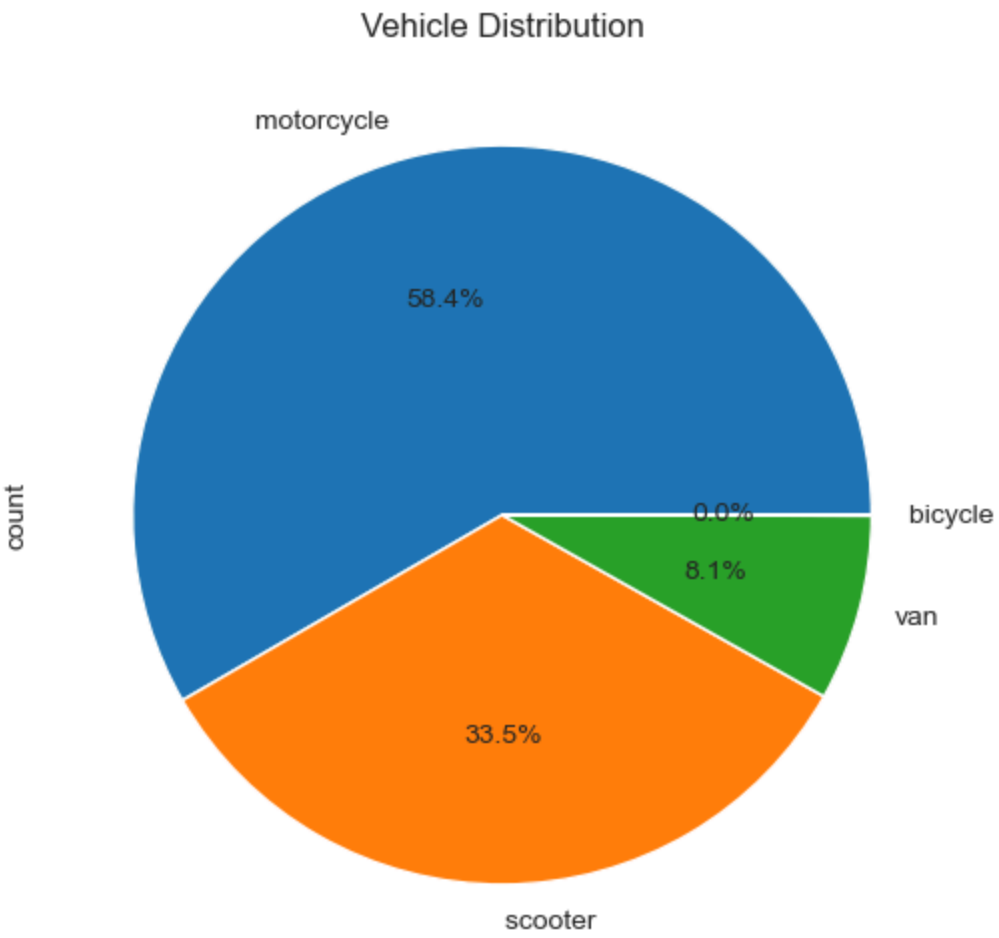
The bar chart illustrates the distribution of orders by area, highlighting significant variations in the volume of orders across four categories: Urban, Metropolitan, Semi-Urban, and Other. The "Metropolitan" area dominates with the highest count of orders, exceeding 30,000, indicating a high concentration of commercial or consumer activity in these regions. "Urban" areas follow with a significantly lower but still substantial number of orders, around 10,000. In contrast, "Semi-Urban" and "Other" categories exhibit minimal order volumes, suggesting limited activity or smaller market sizes in these areas. This distribution emphasizes the economic and commercial prominence of metropolitan regions compared to others, possibly driven by higher population density, infrastructure, or accessibility.

In [76]:
```
# Plot for Category
def plot_category(df):
    plot_distribution('Category', 'Orders by Category')

plot_category(df)
```



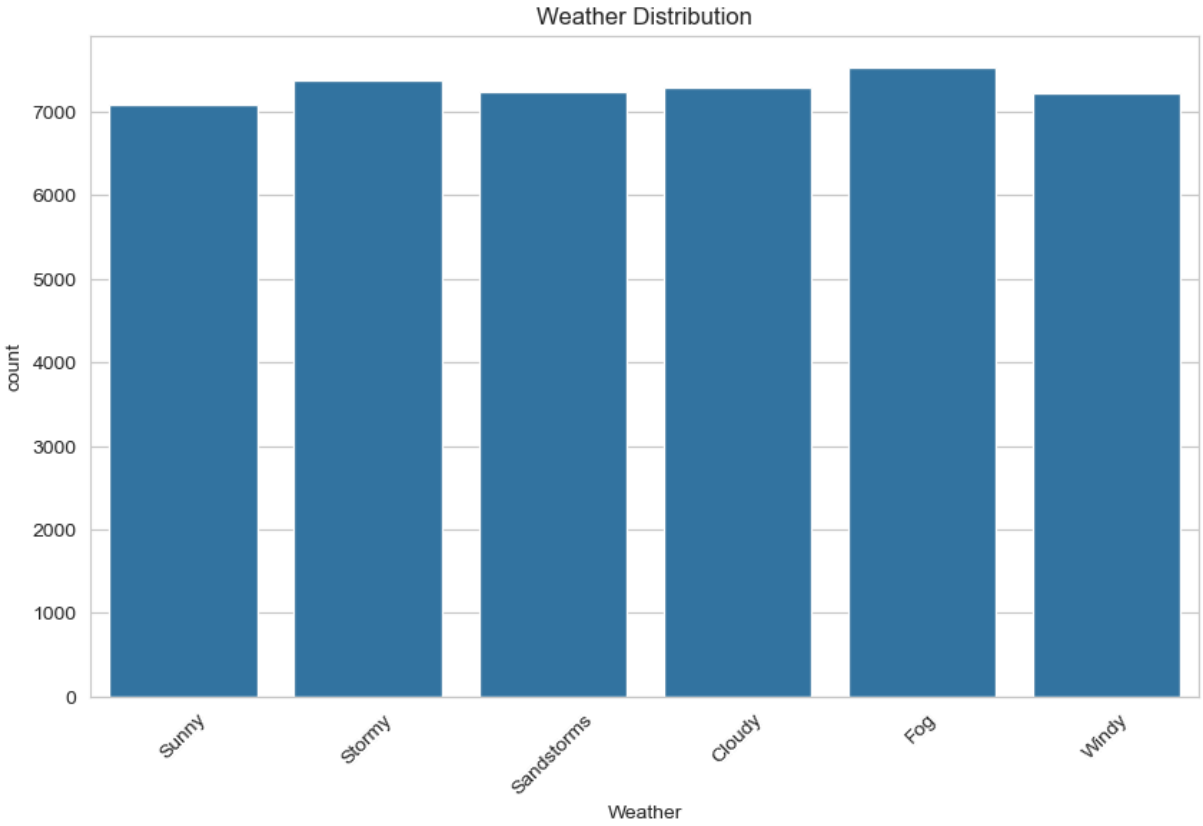
The bar chart represents the distribution of orders across different product categories, indicating a relatively uniform distribution across all categories. Categories such as Clothing, Electronics, Sports, Cosmetics, Groceries, and others have similar order volumes, ranging between approximately 2,500 and 3,000 orders. This suggests a balanced consumer interest and demand across diverse product types without significant dominance by any single category. The uniformity in the distribution could reflect a well-diversified marketplace, where no single category significantly outperforms the others in terms of order count. This information can be useful for inventory planning, marketing strategies, or analyzing overall consumer behavior.

In [77]:
```
# Plot for Vehicle
def  plot_vehicle(df):
    plot_pie('Vehicle', 'Vehicle Distribution')

plot_vehicle(df)
```



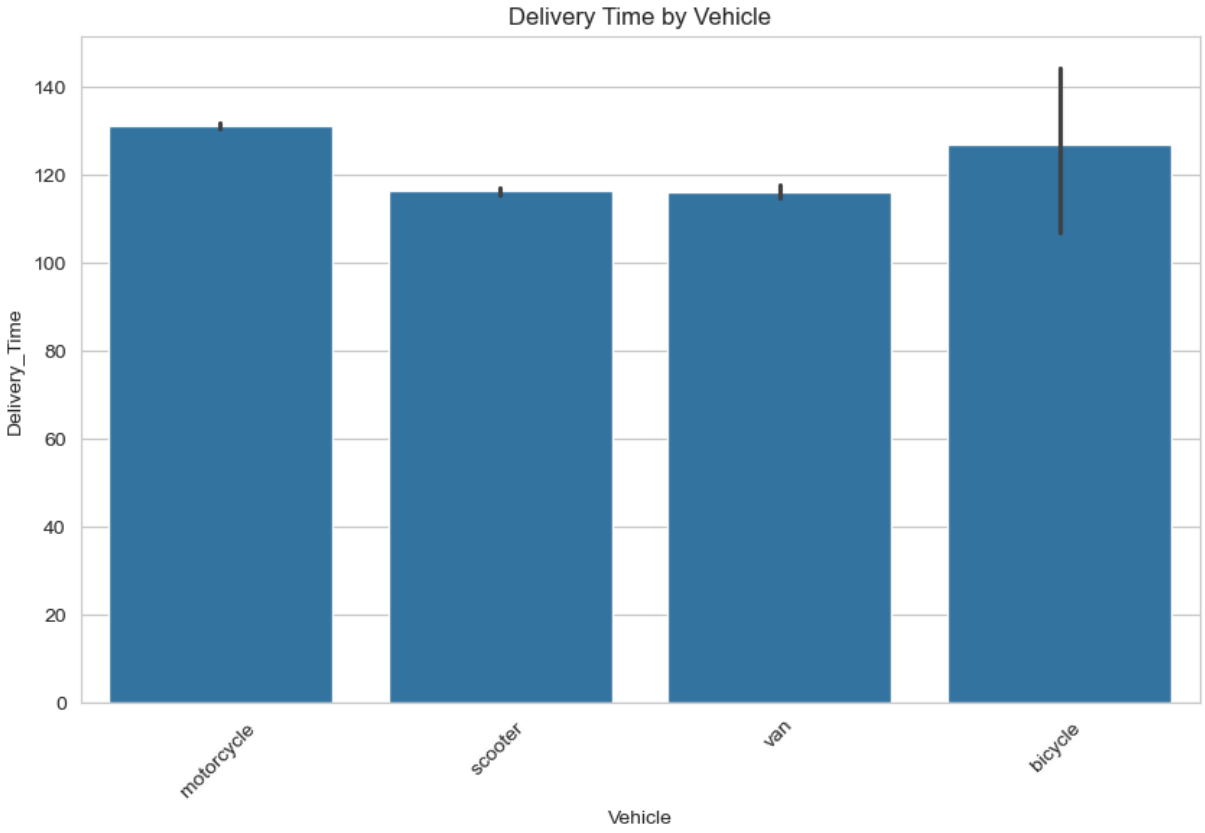
The pie chart illustrates the distribution of vehicle types used for deliveries, highlighting significant disparities in their usage. Motorcycles constitute the majority, accounting for 58.4% of all deliveries, followed by scooters at 33.5%. Vans contribute a smaller share at 8.1%, while bicycles have an almost negligible share, representing 0.0%. This distribution indicates that motorcycles and scooters are the primary modes of transportation for deliveries, likely due to their speed, efficiency, and suitability for navigating urban and metropolitan areas. Vans, while less commonly used, may cater to bulkier or larger deliveries. The minimal use of bicycles suggests they are not a practical option for most delivery operations in this context. This data could inform decisions regarding fleet management and operational optimizations.

In [78]:
```
# Plot for weather distribution
def plot_weather(df):
    plot_distribution('Weather', 'Weather Distribution')

plot_weather(df)
```



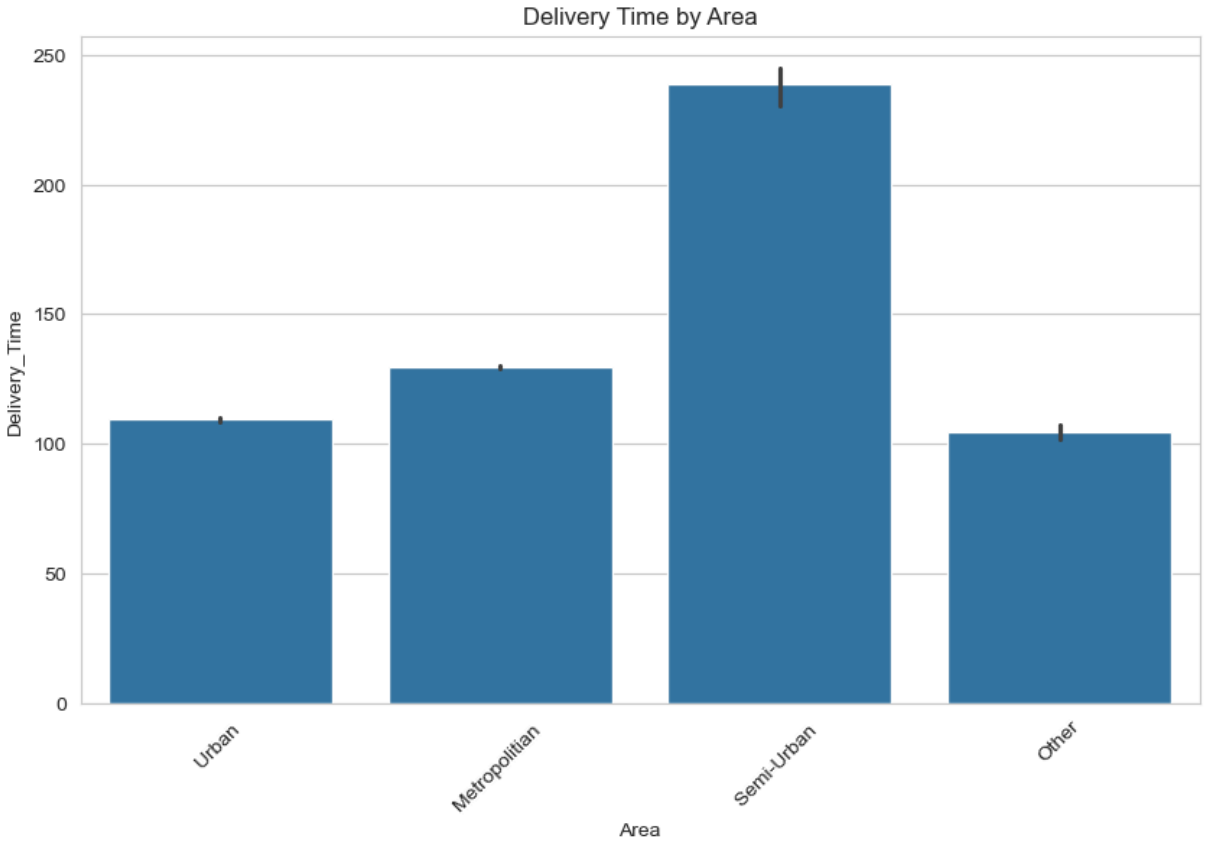
The bar chart illustrates the distribution of delivery counts under various weather conditions, including Sunny, Stormy, Sandstorms, Cloudy, Fog, and Windy. The delivery counts across all weather conditions appear relatively uniform, ranging between approximately 6,500 and 7,000. This uniformity suggests that delivery operations are resilient to different weather conditions, potentially indicating that appropriate logistical measures and vehicle choices are in place to maintain efficiency regardless of the weather. However, slight variations may imply a marginal influence of certain weather conditions on delivery volumes, which could be further analyzed for patterns or operational improvements.

In [80]:
```
def plot_delivery_time_vehicle(df):
    plot_multiple_elements('Vehicle', 'Delivery_Time', 'Delivery Time by Vehicle')

plot_delivery_time_vehicle(df)
```



The bar chart represents the average delivery time for various vehicle types, including motorcycles, scooters, vans, and bicycles. Motorcycles and scooters have relatively similar average delivery times, suggesting they are the most efficient and reliable for quick deliveries. Vans show a slightly longer delivery time, which could be attributed to their usage for bulkier or multiple-item deliveries that require more time to load and unload. Bicycles have the longest average delivery time, reflecting their slower speed and limited suitability for long-distance or high-volume deliveries. These insights can guide fleet optimization by matching vehicle types to delivery requirements based on efficiency.

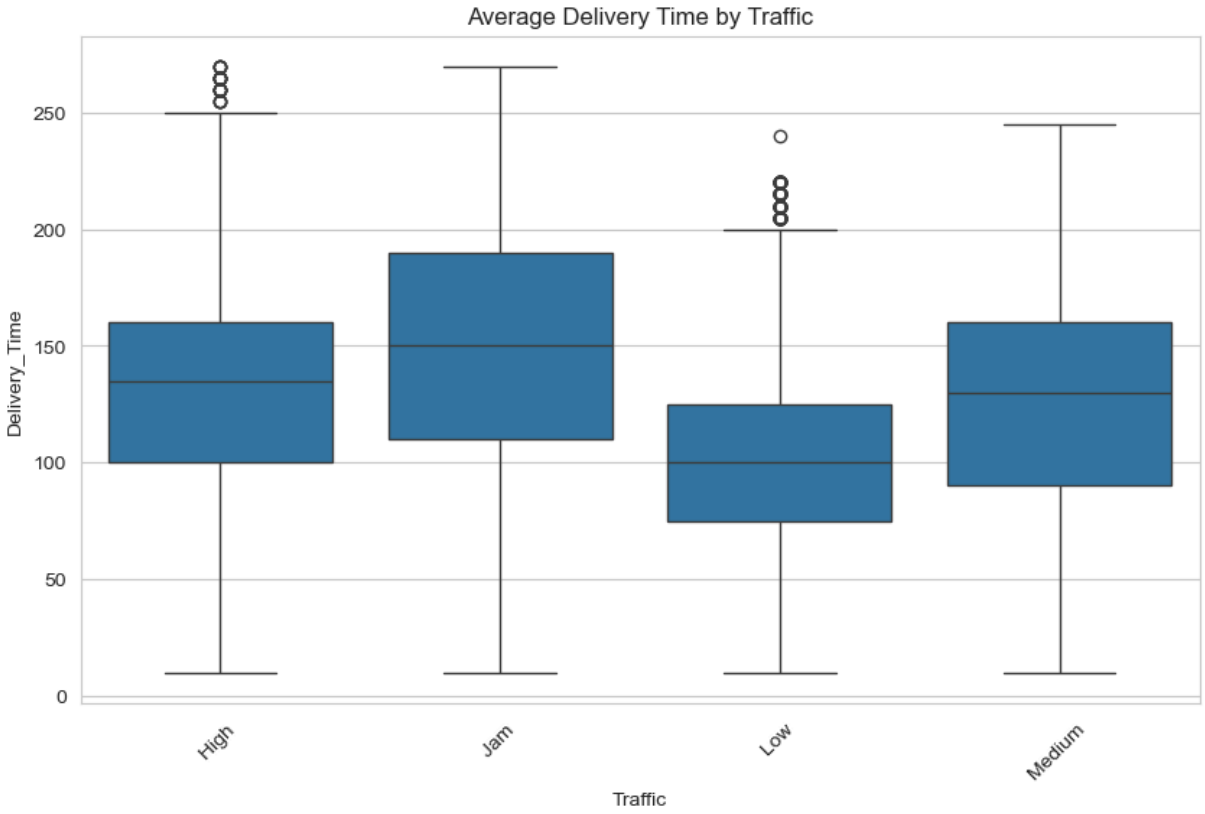
In [81]:
```
# plot for delivery time by area
def plot_delivery_time_area(df):
    plot_multiple_elements('Area', 'Delivery_Time', 'Delivery Time by Area')

plot_delivery_time_area(df)
```



The bar chart illustrates the average delivery time by area, with categories including Urban, Metropolitan, Semi-Urban, and Other. The Semi-Urban area shows the highest average delivery time, exceeding 250 minutes, indicating potential logistical challenges such as greater distances or less accessible routes in these areas. The Urban and Metropolitan areas have similar, significantly lower delivery times, reflecting their dense infrastructure and proximity to delivery hubs. The Other category also shows relatively low delivery times, comparable to Urban and Metropolitan areas. This data highlights the need for targeted strategies to improve delivery efficiency in Semi-Urban areas, such as optimizing routes or deploying more suitable vehicle types.

In [114…

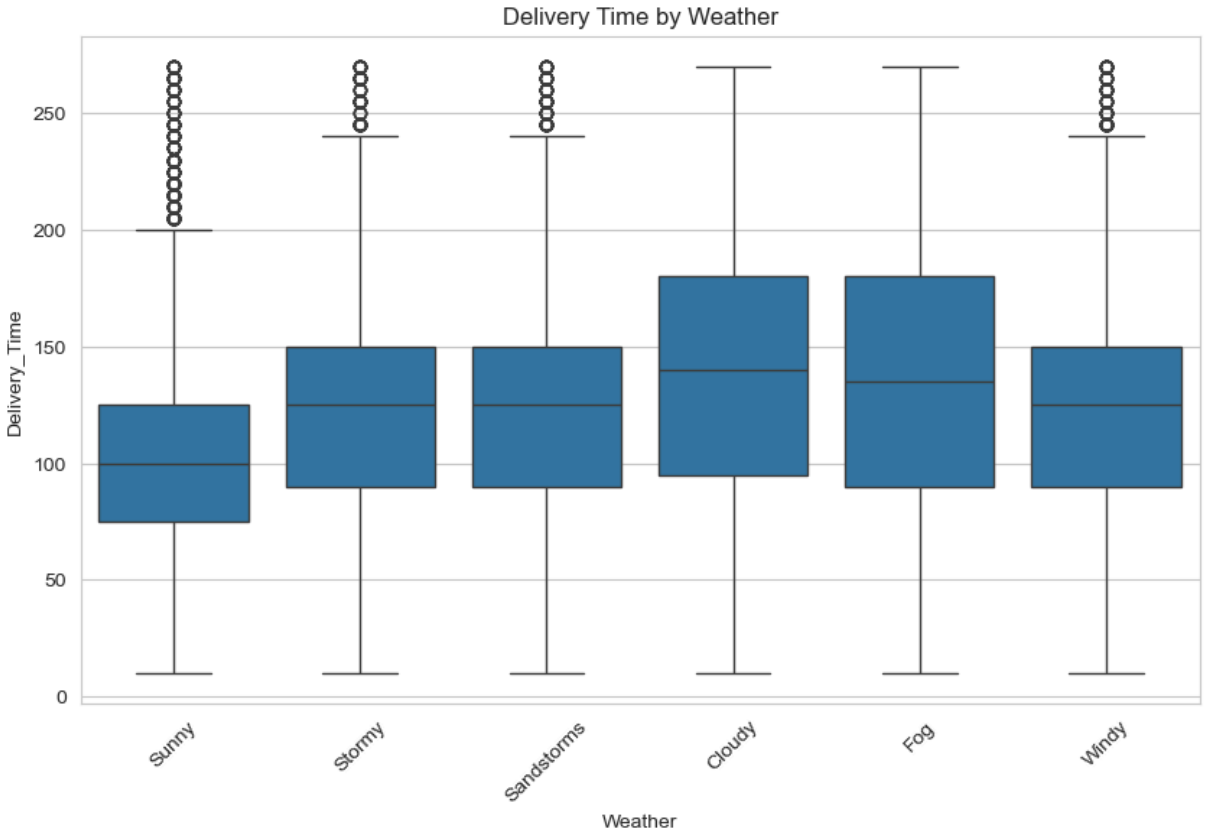
```
# delivery time by traffic
def plot_delivery_time_traffic(df):
    plot_boxplot('Traffic', 'Delivery_Time', 'Average Delivery Time by Traffic')

plot_delivery_time_traffic(df)
```



The boxplot visualizes the distribution of delivery times across different traffic conditions: High, Jam, Low, and Medium. Deliveries during Traffic Jams exhibit the longest delivery times, with a wider spread and more outliers, indicating variability and delays caused by congestion. Medium traffic conditions also show relatively high delivery times, though with less variability. High and Low traffic conditions have shorter average delivery times and smaller spreads, suggesting more consistent delivery performance in these scenarios. The data highlights that traffic jams significantly impact delivery efficiency, emphasizing the importance of route optimization and traffic-aware planning to mitigate delays during peak congestion times.

In [112…

```
# plot for weeather on delivery time
def plot_delivery_time_weather(df):
    plot_boxplot('Weather', 'Delivery_Time', 'Delivery Time by Weather')

plot_delivery_time_weather(df)
```

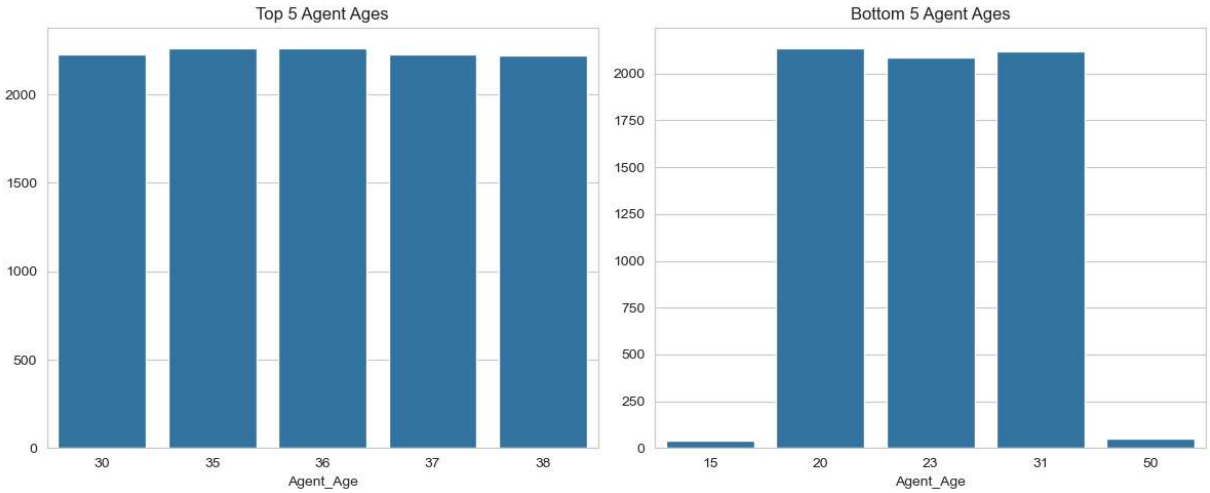


The boxplot visualizes the impact of different weather conditions on Delivery Time. It demonstrates notable variability across weather types. Sunny weather is associated with the lowest delivery times, with a narrower range and fewer outliers, indicating consistent and efficient deliveries. Conversely, stormy and snowy conditions show significantly higher median delivery times and wider interquartile ranges, suggesting increased delays and variability under adverse weather conditions. Cloudy, foggy, and windy weather exhibit moderate delays, with their medians slightly above that of sunny weather but below stormy and snowy conditions. The results emphasize the influence of weather on delivery efficiency, with adverse weather conditions such as storms and snow being major contributors to delays. This insight could guide logistics planning, encouraging proactive measures like adjusting routes or allocating additional time buffers during challenging weather conditions.

In [97]:
```
# plot to show agent ages of top 5 and bottom 5
def plot_agent_ages(df):
    plt.figure(figsize=(12, 5))

    # Created two subplots
    # The first subplot shows the top 5 agent ages
    # The second subplot shows the bottom 5 agent ages

    plt.figure(figsize=(12, 5))

    # Subplot 1: Top 5 agent ages
    plt.subplot(1, 2, 1)
    top_ages = df['Agent_Age'].value_counts().head()
    sns.barplot(x=top_ages.index, y=top_ages.values)
```

```
    plt.title('Top 5 Agent Ages')

    # Subplot 2: Bottom 5 agent ages
    plt.subplot(1, 2, 2)
    bottom_ages = df['Agent_Age'].value_counts().tail()
    sns.barplot(x=bottom_ages.index, y=bottom_ages.values)
    plt.title('Bottom 5 Agent Ages')

    plt.tight_layout()
    plt.show()


plot_agent_ages(df)
```

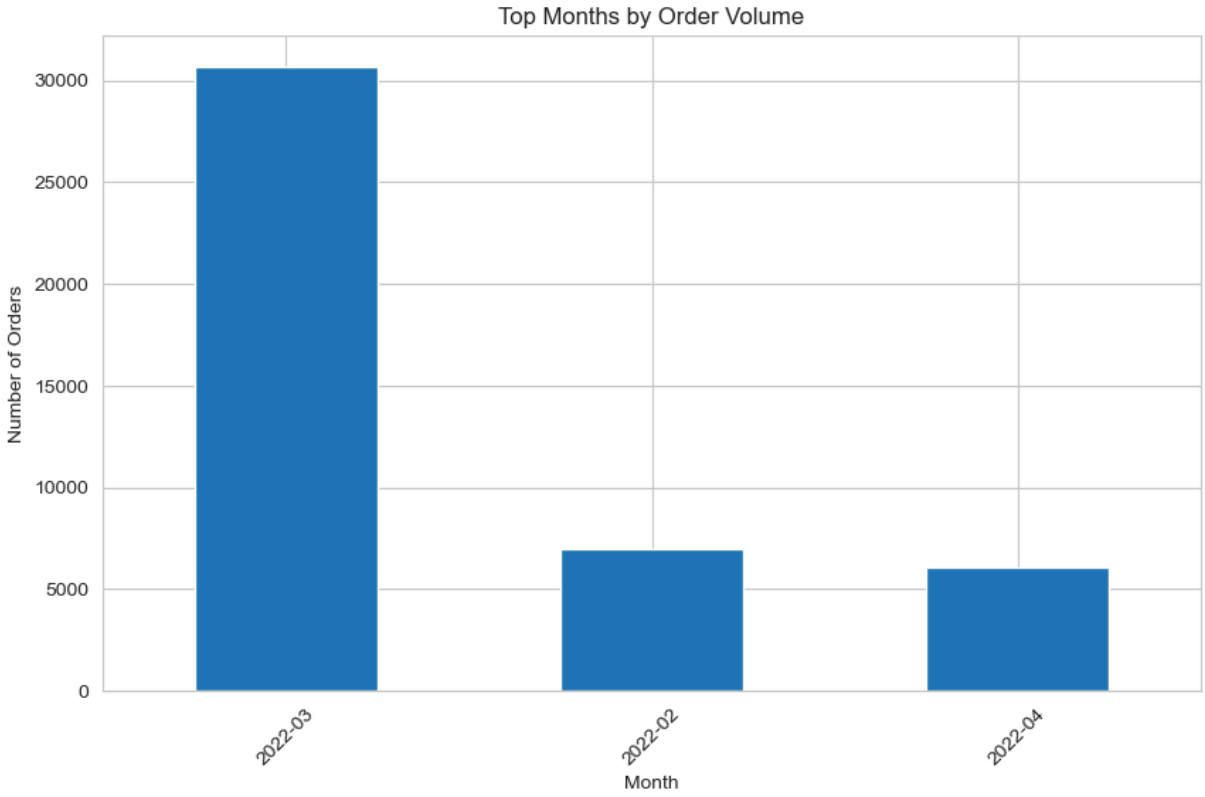
```
<Figure size 1200x500 with 0 Axes>
```



The bar charts depict the top 5 agent ages and the bottom 5 agent ages based on order counts. Among the top-performing agents, ages range between 30 and 38, with each age group contributing roughly similar numbers of orders, all exceeding 2,000. This suggests that agents within this age range are likely the most active or efficient in handling deliveries. Conversely, the bottom-performing agents are younger, between ages 15 and 23, handling significantly fewer orders, all below 500. This discrepancy may point to factors such as experience, physical capability, or availability influencing agent performance. These insights could inform recruitment and training strategies, focusing on leveraging the strengths of more experienced agents while identifying areas for development among younger agents.

In [99]:
```
# plot for monthly orders
def plot_monthly_orders(df):
    # Converted Order_Date to datetime
    df['Order_Date'] = pd.to_datetime(df['Order_Date'])

    # Extracted month and year
    monthly_orders = df.groupby(
        df['Order_Date'].dt.strftime('%Y-%m')).size().sort_values(ascending=False)

    plt.figure()
    monthly_orders.head().plot(kind='bar')
    plt.title('Top Months by Order Volume')
    plt.xlabel('Month')
```

```
    plt.ylabel('Number of Orders')
    plt.xticks(rotation=45)
    plt.show()


plot_monthly_orders(df)
```

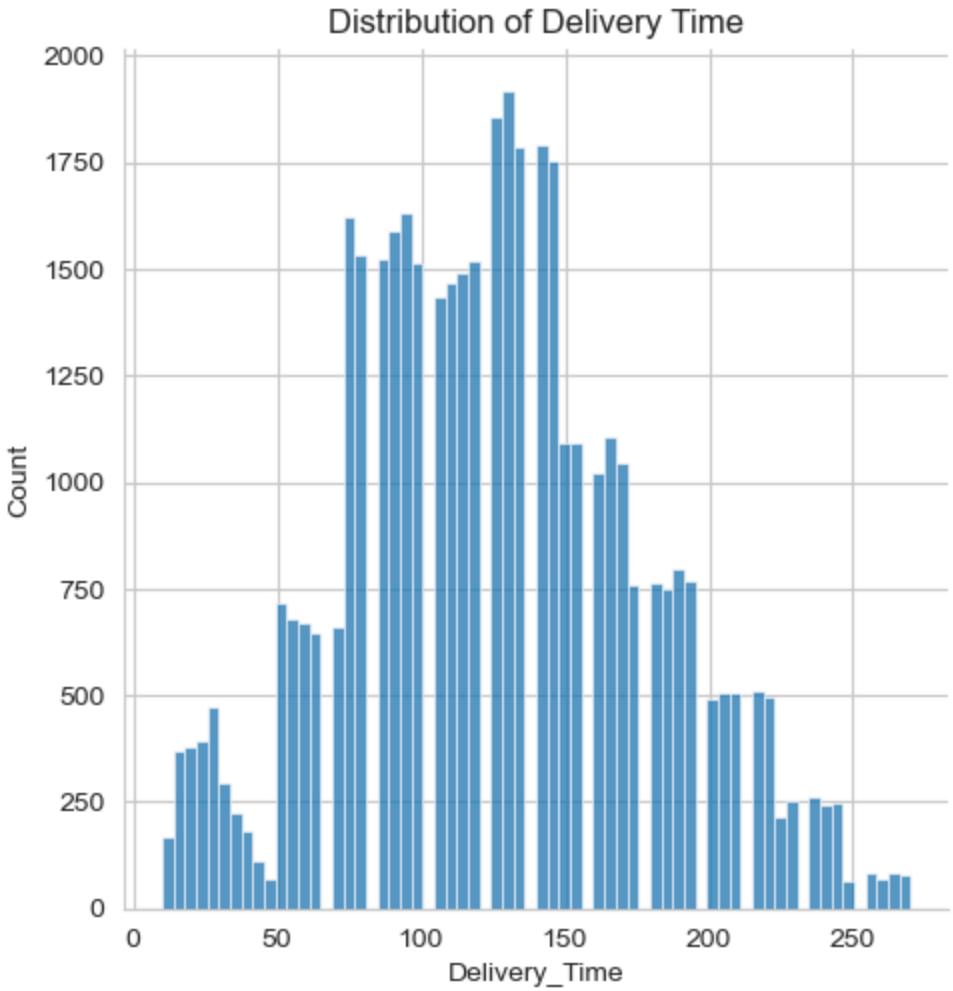


The bar chart displays the top months by order volume, highlighting significant differences in monthly sales performance. The month March 2022 records the highest number of orders, exceeding 30,000, indicating a peak in business activity during this period. In contrast, February 2022 and April 2022 exhibit substantially lower order volumes, with both months registering less than half the orders seen in March. This stark disparity suggests the presence of a seasonal or promotional effect in March, which could be attributed to events like sales campaigns, holidays, or other demand-driving factors. Businesses could leverage this insight by targeting similar periods for marketing and operational scaling to optimize order fulfillment.

In [115…

```
# plot for delivery time
def plot_delivery_time(df):
    plt.figure()
    sns.displot(df, x='Delivery_Time', kind="hist")
    plt.title('Distribution of Delivery Time')
    plt.show()

plot_delivery_time(df)
```

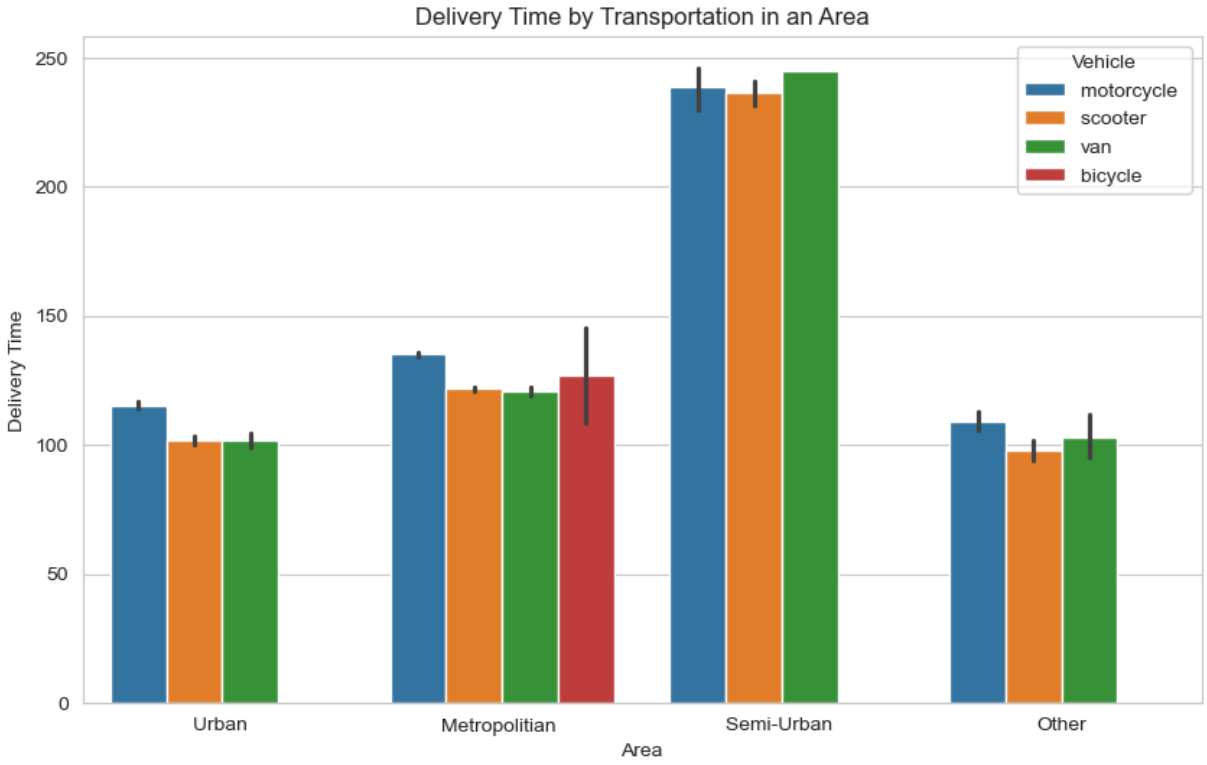
```
<Figure size 1000x600 with 0 Axes>
```



The histogram represents the distribution of delivery times, showcasing a range of durations with notable peaks and a skewed spread. The majority of deliveries occur within the 50 to 150-minute range, with a peak around 100 minutes, indicating this is the most common delivery duration. Delivery times extend up to 250 minutes, but the frequency decreases as the time increases, reflecting fewer deliveries with long durations. This distribution suggests that most deliveries are completed within a reasonable time frame, while a smaller proportion takes longer, likely due to external factors such as traffic congestion, route inefficiencies, or challenging delivery areas. Understanding this distribution can help identify bottlenecks and optimize processes for quicker deliveries.

In [141…

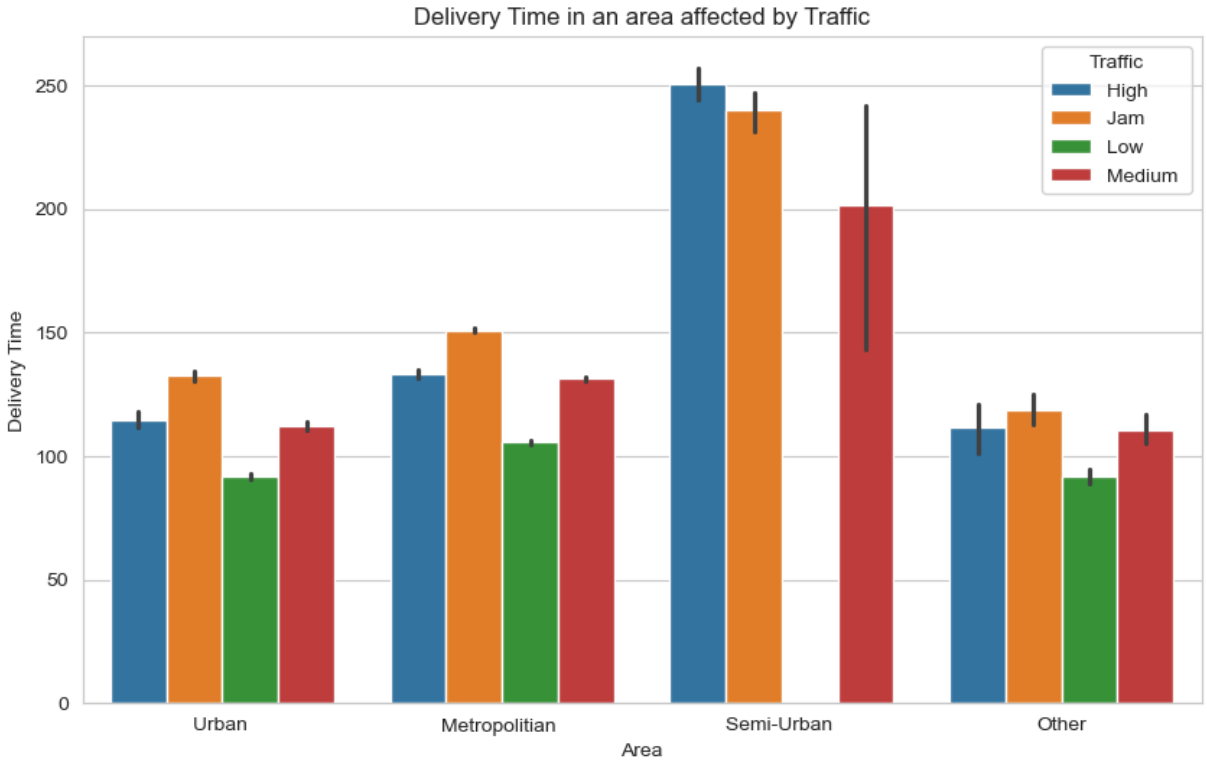
```
#plot for delivery time by transportation in an area
# this graph shows the delivery times by different transportations in an area(urban
def plot_delivery_by_transportation_in_area(df):
    plt.figure()
    sns.barplot(data=df, x='Area', y='Delivery_Time', hue= 'Vehicle')
    plt.title('Delivery Time by Transportation in an Area')
    plt.xlabel('Area')
    plt.ylabel('Delivery Time')
    plt.show()

plot_delivery_by_transportation_in_area(df)
```



The grouped bar chart compares delivery times by transportation mode (motorcycle, scooter, van, bicycle) across different areas (Urban, Metropolitan, Semi-Urban, and Other). Semi-Urban areas consistently show the longest delivery times across all transportation modes, highlighting logistical challenges in these regions. In contrast, Urban and Metropolitan areas have shorter and relatively consistent delivery times for all vehicle types, reflecting the efficiency of these densely populated and better-connected areas. Bicycles exhibit the highest delivery times across all regions, while motorcycles and scooters generally perform better, emphasizing their suitability for quicker deliveries. Vans, while slower than two-wheelers, maintain moderate delivery times, likely attributed to their capacity for bulk deliveries. These insights suggest a need to optimize logistics in Semi-Urban areas and consider vehicle-specific strategies for improving delivery efficiency.

In [142…

```
#plot for delivery time by transportation in an area affected by trafffic
def plot_delivery_in_area_for_traffic(df):
    plt.figure()
    sns.barplot(data=df, x='Area', y='Delivery_Time', hue= 'Traffic')
    plt.title('Delivery Time in an area affected by Traffic')
    plt.xlabel('Area')
    plt.ylabel('Delivery Time')
    plt.show()

plot_delivery_in_area_for_traffic(df)
```

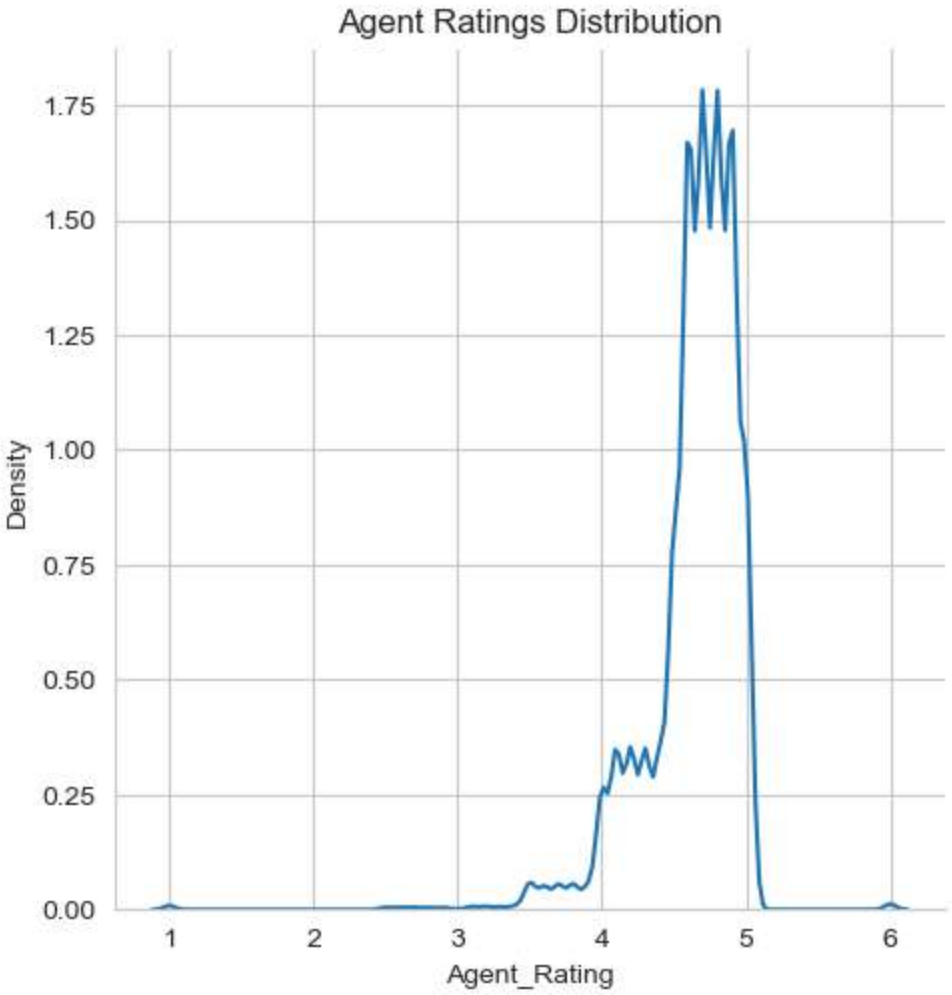


The bar chart illustrates the impact of traffic conditions (categorized as High, Jam, Low, and Medium) on Delivery Time across different areas (Urban, Metropolitan, Semi-Urban, and Other). It is evident that areas experiencing high traffic congestion and jams consistently show significantly higher delivery times compared to low or medium traffic conditions. Semi-Urban and "Other" areas are most affected by high traffic, with delivery times exceeding 250 minutes, while Urban areas demonstrate the shortest delivery times under all traffic conditions, likely due to better infrastructure or proximity between delivery points. The chart highlights the critical role of traffic conditions in influencing delivery efficiency, with implications for optimizing route planning, especially in highly congested regions like Semi-Urban and Other areas.

In [143…

```
# Plot for Agent Ratings
def plot_agent_rating(df):
    plt.figure()

    # kde means kernel density estimation or kernel density plot, which is a way to
    # probability density function of a random variable in this case the Agent_Rati
    sns.displot(data=df, x='Agent_Rating', kind='kde')

    plt.title('Agent Ratings Distribution')
    plt.show()

plot_agent_rating(df)
```

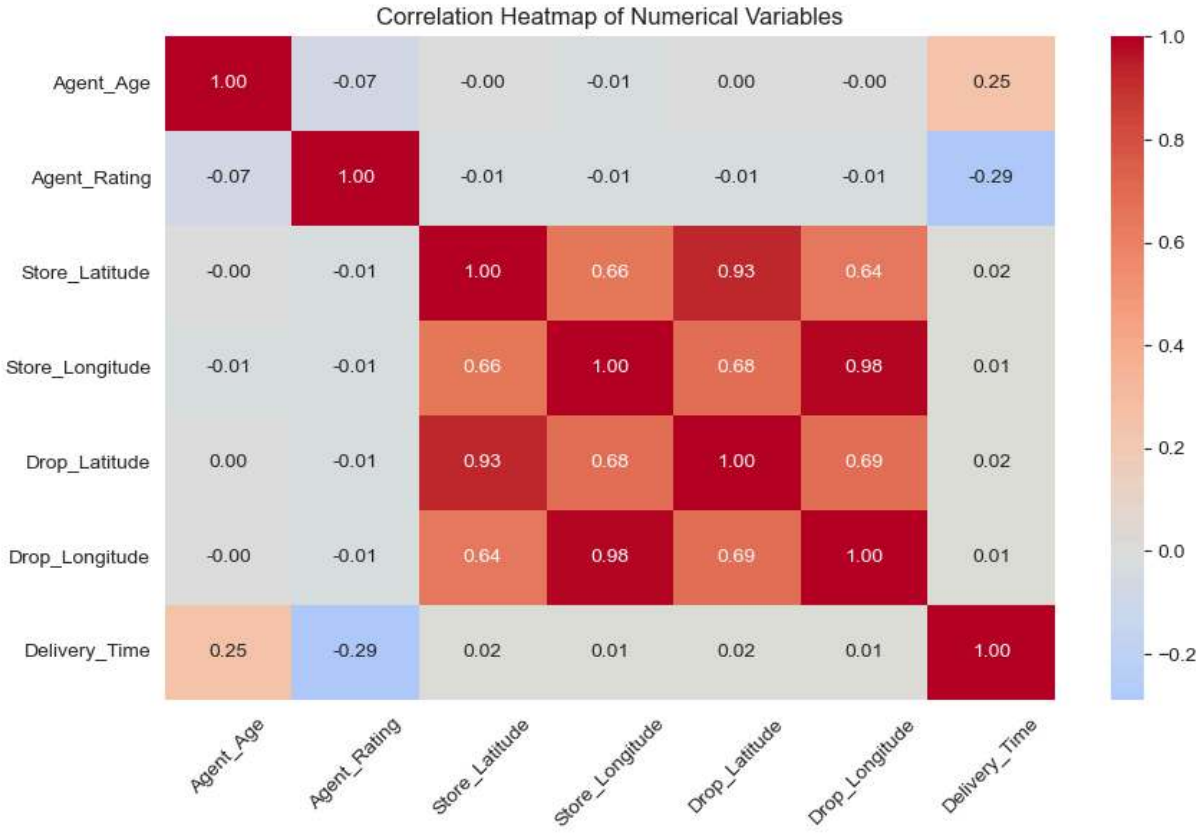
```
<Figure size 1000x600 with 0 Axes>
```



The density plot illustrates the distribution of Agent Ratings, which appears to be highly skewed towards the upper end of the rating scale, predominantly between 4.5 and 5. This indicates that most agents are rated highly, suggesting good performance or customer satisfaction. There is minimal variability in lower rating ranges (below 4), reflecting that poor ratings are uncommon. This skewed distribution could lead to a lack of variability in predictions when using Agent Ratings as a feature in machine learning models. The concentration of ratings near 5 also raises the question of whether the rating system may suffer from a ceiling effect, where higher ratings are capped and fail to capture meaningful differentiation between agents' performances. Further investigation into the rating process might be necessary to determine whether this bias is systemic.

In [ ]:
```
# Plot for correlation heatmap
def plot_correlation_heatmap(df):

    # We already identified the numerical columns earlier
    correlation_matrix = df[numerical_columns].corr()

    plt.figure()
    sns.heatmap(correlation_matrix,
                annot=True,
                cmap='coolwarm',
                center=0,
```

```
                fmt='.2f')
    plt.title('Correlation Heatmap of Numerical Variables')
    plt.xticks(rotation=45)
    plt.show()

plot_correlation_heatmap(df)
```



The heatmap illustrates the correlation between numerical variables in the dataset. It reveals strong positive correlations between geographical variables, such as Store_Latitude and Store_Longitude (0.98), and Store_Latitude and Drop_Latitude (0.93), suggesting these variables are closely related spatially. Similarly, Store_Longitude and Drop_Longitude are highly correlated (0.98), reflecting proximity between stores and drop locations. However, Agent_Age and Agent_Rating exhibit weak correlations with all variables, indicating minimal influence on other factors. Interestingly, Delivery_Time shows low correlations across all features, with its highest correlation being with Agent_Rating (-0.29) and Agent_Age (0.25), suggesting that delivery time is only slightly affected by these attributes. These findings imply that geographical variables are interdependent, while agent-related factors and delivery time are weakly related to the rest of the variables, highlighting potential areas for further investigation or feature engineering in predictive modeling.

5. Machine Learning Models

Machine learning models are the cornerstone of predictive analysis, which enables the discovery of patterns and insights that are not easily identifiable through traditional

methods. In this project, machine learning is utilized to predict delivery times accurately and to evaluate the influence of factors like agent information, traffic, weather, vehicle type, and area on delivery efficiency.

Random Forest Model

Random forest model is a versatile and ensemble learning algorithm which helps to improve accuracy and reduce the risk of overfitting.

A model to predict the delivery time of an order affected by several factors such as location, weather, time and type of vehicle. The model will show the impact of each factor on the delivery time and wheather the delivery time will be overestimated or underestimated.

In [147…

```
# Importing the libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_absolute_percentage_error
import matplotlib.pyplot as plt
```

Data Preprocessing of data

In [163…

```
# Starting by preprocessing the data
def preprocess_data(df, selected_columns):
    df_processed = df[selected_columns].copy()

    # Handle time columns
    # Extract hours from Order_Time and Pickup_Time and convert to integers
    if 'Order_Time' in df_processed.columns:
        # Extract hour directly from the time string
        df_processed['Order_Hour'] = df_processed['Order_Time'].str.split(':').str[
        df_processed = df_processed.drop('Order_Time', axis=1)

    if 'Pickup_Time' in df_processed.columns:
        # Extract hour directly from the time string
        df_processed['Pickup_Hour'] = df_processed['Pickup_Time'].str.split(':').st
        df_processed = df_processed.drop('Pickup_Time', axis=1)

    # Handle categorical columns
    for col in categorical_columns:
        if col in df_processed.columns:
            encoder = LabelEncoder()
            df_processed[col] = encoder.fit_transform(df_processed[col])

    # Scale numerical features
    # StandardScaler is a preprocessing step that scales the
    # features to have a mean of 0 and a standard deviation of 1
    scaler = StandardScaler()
```

```
    df_processed = pd.DataFrame(
        scaler.fit_transform(df_processed),
        columns=df_processed.columns
    )

    print("Data preprocessing completed.")
    return df_processed
```

Model Training and Evaluation

In [167…

```
# Next is to train the models and evaluate their performance.
# The performance shows the accuracy of the model and displays the results.

def train_model(model, X, y):
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    mape = mean_absolute_percentage_error(y_test, predictions)

    # Calculate accuracy, which is 100% minus the MAPE
    # MAPE is a measure of how accurate the model is.
    accuracy = 100 * (1 - mape)

    return {
        'model': model,
        'actual': y_test,
        'predicted': predictions,
        'accuracy': accuracy,
        'X_test': X_test
    }

print("Training and evaluation complete.")
```

```
Training and evaluation complete.
```

In [183…

```
# Next is to run the machine learning models and see the results and accuracy.
#n_estimators mean the number of trees in the model, which is simple terms  the num

# Function to create a random forest model
def random_forest_model():
    print("Random Forest Model sucess.")
    return RandomForestRegressor(n_estimators=100, random_state=42)

# Function to create a gradient boosting model
def gradient_boosting_modelling():
    print("Gradient Boosting Model sucess.")
    return GradientBoostingRegressor(n_estimators=100, random_state=42)

# Function to create a polynomial regression model
def polynomial_regression_model(degree=2):
    poly_features = PolynomialFeatures(degree=degree)
```

```
    linear_regression = LinearRegression()

    # The Poly_model class is a wrapper around the PolynomialFeatures and LinearReg
    # It allows for the creation of a polynomial regression model with a specified
    class Poly_model:
        def __init__(self, poly_features, linear_regression):
            self.poly_features = poly_features
            self.linear_regression = linear_regression

        def fit(self, X, y):
            X_poly = self.poly_features.fit_transform(X)
            self.linear_regression.fit(X_poly, y)
            return self

        def predict(self, X):
            X_poly = self.poly_features.transform(X)
            return self.linear_regression.predict(X_poly)

    print("Polynomial Regression Model sucess.")
    return Poly_model(poly_features, linear_regression)
```

In [174…

```
# Next is to display the accuracy, results and plot the predictions.

# Function to display accuracy
def display_accuracy(model_results):
    print(f"\nModel Accuracy: {model_results['accuracy']:.2f}%")

# Function to display the results
def display_prediction_examples(model_results, num_examples=5):
    print("\nPrediction Examples")


    for i in range(min(num_examples, len(model_results['actual']))):
        actual = model_results['actual'].iloc[i]
        predicted = model_results['predicted'][i]
        diff = predicted - actual
        error_percent = (abs(diff) / actual) * 100

        print(f"\nDelivery {i+1}:")
        print(f"Actual Time: {actual:.0f} minutes")
        print(f"Predicted Time: {predicted:.0f} minutes")
        print(f"Difference: {diff:.0f} minutes")
        print(f"Error: {error_percent:.1f}%")
        print(f"Delivery time was {'underestimated' if diff < 0 else 'overestimated
```

In [154…

```
# Next steps is to plot the predictions and feature importance for each model.

def plot_results(model_results, model_name):
    plt.figure(figsize=(12, 6))

    # Scatter plot to show actual vs predicted values
    plt.scatter(model_results['actual'], model_results['predicted'],
                alpha=0.5, label='Predictions')

    # Adding a line for perfect prediction, which in simple terms means the line of
```

```
    min_val = min(model_results['actual'].min(), model_results['predicted'].min())
    max_val = max(model_results['actual'].max(), model_results['predicted'].max())
    plt.plot([min_val, max_val], [min_val, max_val],
             'r--', label='Perfect Prediction')

    # Adding labels and title for the plot
    plt.xlabel('Actual Delivery Time (minutes)')
    plt.ylabel('Predicted Delivery Time (minutes)')
    plt.title(f'{model_name} Prediction Results\nAccuracy: {model_results["accuracy
    plt.legend()
```

In [170…

```
# Selecting the columns and target columns to be used in the models

selected_columns = ['Agent_Age', 'Agent_Rating', 'Store_Latitude', 'Store_Longitude
           'Drop_Latitude', 'Drop_Longitude', 'Order_Time', 'Pickup_Time',
           'Weather', 'Traffic', 'Vehicle', 'Area', 'Category']

X = df[selected_columns]

#Target column
y = df['Delivery_Time']

# Preprocess the dataset
processed_dataset = preprocess_data(X, selected_columns)
```

```
Data preprocessing completed.
```

Random Forest model

In [ ]:

```
# Train the Random Forest model
random_forest = random_forest_model()
random_forest = train_model(random_forest, processed_dataset, y)
```

```
Random Forest Model sucess.
```

In [172…

```
#Get the accuracy of the model
display_accuracy(random_forest)
```

```
Model Accuracy: 80.49%
```

In [175…

```
#Display results of the model
display_prediction_examples(random_forest)
```

```
Prediction Examples

Delivery 1:
Actual Time: 145 minutes
Predicted Time: 109 minutes
Difference: -36 minutes
Error: 24.7%
Delivery time was underestimated

Delivery 2:
Actual Time: 21 minutes
Predicted Time: 26 minutes
Difference: 5 minutes
Error: 24.0%
Delivery time was overestimated

Delivery 3:
Actual Time: 165 minutes
Predicted Time: 122 minutes
Difference: -43 minutes
Error: 26.3%
Delivery time was underestimated

Delivery 4:
Actual Time: 200 minutes
Predicted Time: 177 minutes
Difference: -23 minutes
Error: 11.7%
Delivery time was underestimated

Delivery 5:
Actual Time: 145 minutes
Predicted Time: 155 minutes
Difference: 10 minutes
Error: 7.0%
Delivery time was overestimated
```

Observation: Delivery 1: The model underestimated the delivery time by 36 minutes, resulting in a high error of 24.7%. Such a significant underestimation could negatively impact customer expectations. Delivery 2: The model overestimated the delivery time by 5 minutes, with an error of 24.0%. Although the absolute difference is small, the percentage error is large due to the short delivery time. Delivery 3: The model underestimated the delivery time by 43 minutes, with an error of 26.3%. This large underestimation indicates that the model struggles with predicting longer delivery times accurately. Delivery 4: The model underestimated the delivery time by 23 minutes, but the error is relatively low at 11.7%. This suggests that the model performs better for moderately long delivery times. Delivery 5: The model overestimated the delivery time by 10 minutes, resulting in a low error of 7.0%. This is the best performance among the five cases, indicating the model can achieve good accuracy under certain conditions.

Gradient Boosting Model

Gradient boosting model builds a predictive model incrementally from a series of weak learners, typically decision trees. Unlike other ensemble methods, Gradient Boosting optimizes the model performance by minimizing errors in a step-by-step manner through gradient descent.

In [184…
```
#Train the gradient boosting model
gradient_boosting = gradient_boosting_modelling()
gradient_boosting = train_model(gradient_boosting, processed_dataset, y)
```

```
Gradient Boosting Model sucess.
```

In [185…
```
#display accuracy of the model
display_accuracy(gradient_boosting)
```

```
Model Accuracy: 78.37%
```

In [186…
```
#display the  results of the model
display_prediction_examples(gradient_boosting)
```

```
Prediction Examples

Delivery 1:
Actual Time: 145 minutes
Predicted Time: 119 minutes
Difference: -26 minutes
Error: 18.0%
Delivery time was underestimated

Delivery 2:
Actual Time: 21 minutes
Predicted Time: 34 minutes
Difference: 13 minutes
Error: 60.2%
Delivery time was overestimated

Delivery 3:
Actual Time: 165 minutes
Predicted Time: 129 minutes
Difference: -36 minutes
Error: 21.7%
Delivery time was underestimated

Delivery 4:
Actual Time: 200 minutes
Predicted Time: 189 minutes
Difference: -11 minutes
Error: 5.5%
Delivery time was underestimated

Delivery 5:
Actual Time: 145 minutes
Predicted Time: 155 minutes
Difference: 10 minutes
Error: 6.6%
Delivery time was overestimated
```

Observation: Delivery 1: The model underestimated the delivery time by 26 minutes, resulting in an error of 18.0%. This suggests the model performs reasonably well but struggles to precisely capture some influencing factors. Delivery 2: The model overestimated the delivery time by 13 minutes, leading to a very high error of 60.2%. This significant error indicates that the model may not handle short delivery times effectively, possibly due to insufficient data or feature limitations for such cases. Delivery 3: The model underestimated the delivery time by 36 minutes, with an error of 21.7%. While the error is moderate, it highlights the need for better predictions for longer delivery times. Delivery 4: The model underestimated the delivery time by 11 minutes, achieving a low error of 5.5%. This is a strong performance, indicating that the model can accurately predict moderately long delivery times. Delivery 5: The model overestimated the delivery time by 10 minutes, resulting in a low error of 6.6%. This is another instance of good performance, with minimal deviation from the actual time.

Polynomial Regression

Polynomial Regression extends the Linear Regression model, adding the relationship of the independent variables to the dependent variable. Contrary to linear regression, assuming that it is a relationship in a straight line, Polynomial Regression can explain more complex relations, nonlinear with the inclusion of higher-degree terms on its features.

In [187…
```
#train polynomial regression model
polynomial_regress = polynomial_regression_model()
polynomial_regress = train_model(polynomial_regress, processed_dataset, y)
```

```
Polynomial Regression Model sucess.
```

In [188…
```
#display the accuracy of the model
display_accuracy(polynomial_regress)
```

```
Model Accuracy: 54.54%
```

In [189…
```
#display the results of the model
display_prediction_examples(polynomial_regress)
```

```
Prediction Examples

Delivery 1:
Actual Time: 145 minutes
Predicted Time: 116 minutes
Difference: -29 minutes
Error: 20.3%
Delivery time was underestimated

Delivery 2:
Actual Time: 21 minutes
Predicted Time: 122 minutes
Difference: 101 minutes
Error: 482.9%
Delivery time was overestimated

Delivery 3:
Actual Time: 165 minutes
Predicted Time: 170 minutes
Difference: 5 minutes
Error: 2.9%
Delivery time was overestimated

Delivery 4:
Actual Time: 200 minutes
Predicted Time: 178 minutes
Difference: -22 minutes
Error: 11.2%
Delivery time was underestimated

Delivery 5:
Actual Time: 145 minutes
Predicted Time: 104 minutes
Difference: -41 minutes
Error: 28.2%
Delivery time was underestimated
```

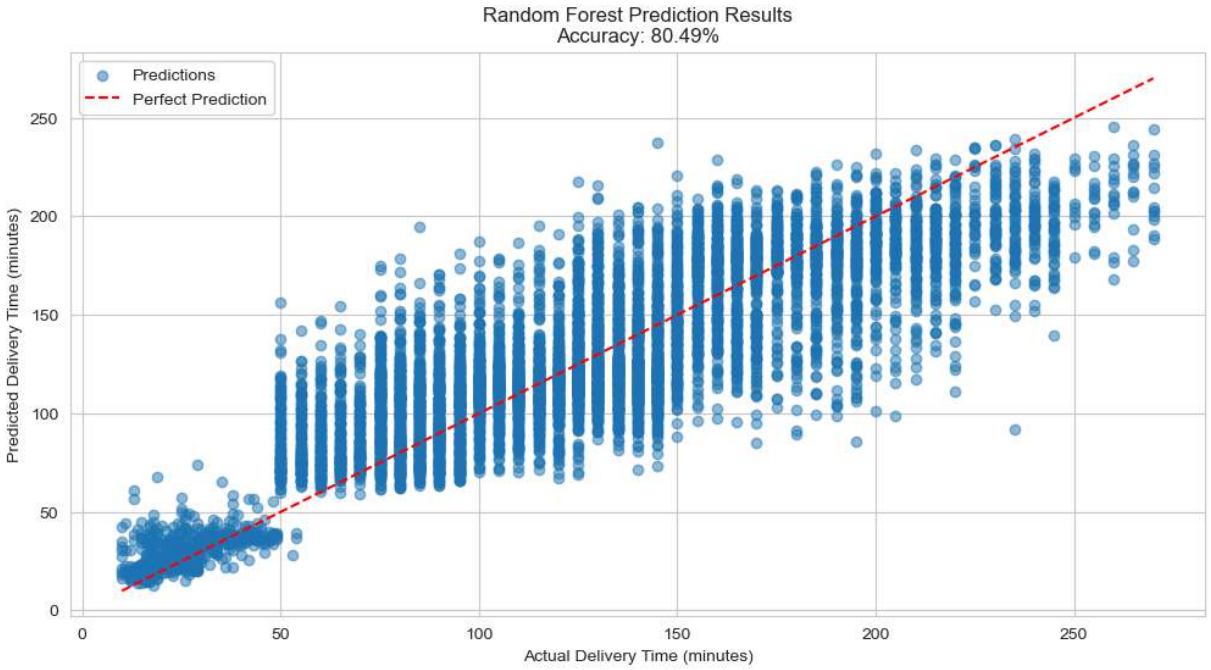
Observation: Delivery 1: The predicted time was lower than the actual time, leading to an underestimate. The difference was -29 minutes (meaning the prediction was too short by 29 minutes). The error is calculated as the percentage difference between the actual and predicted times, which comes out to 20.3%. This means the predicted time was off by 20.3% less than what it actually took. Delivery 2: The predicted time was far higher than the actual time, causing a massive overestimate. The prediction was off by 101 minutes, or 482.9% overestimated. This means the prediction was nearly 5 times longer than it actually took to deliver, making it a significant error. Delivery 3: In this case, the predicted time was just slightly higher than the actual time, which results in a small overestimate. The difference is only 5 minutes, and the error percentage is quite low, at 2.9%. This is a small error, meaning the prediction was fairly accurate.Delivery 4: Again, the predicted time was lower than the actual time, meaning the delivery time was underestimated. The difference is -22 minutes, indicating that the prediction was too short. The error of 11.2% shows that the prediction was off by about 11.2% lower than the actual time. Delivery 5: This delivery was also underestimated, with the predicted time being much lower than the actual delivery time. The

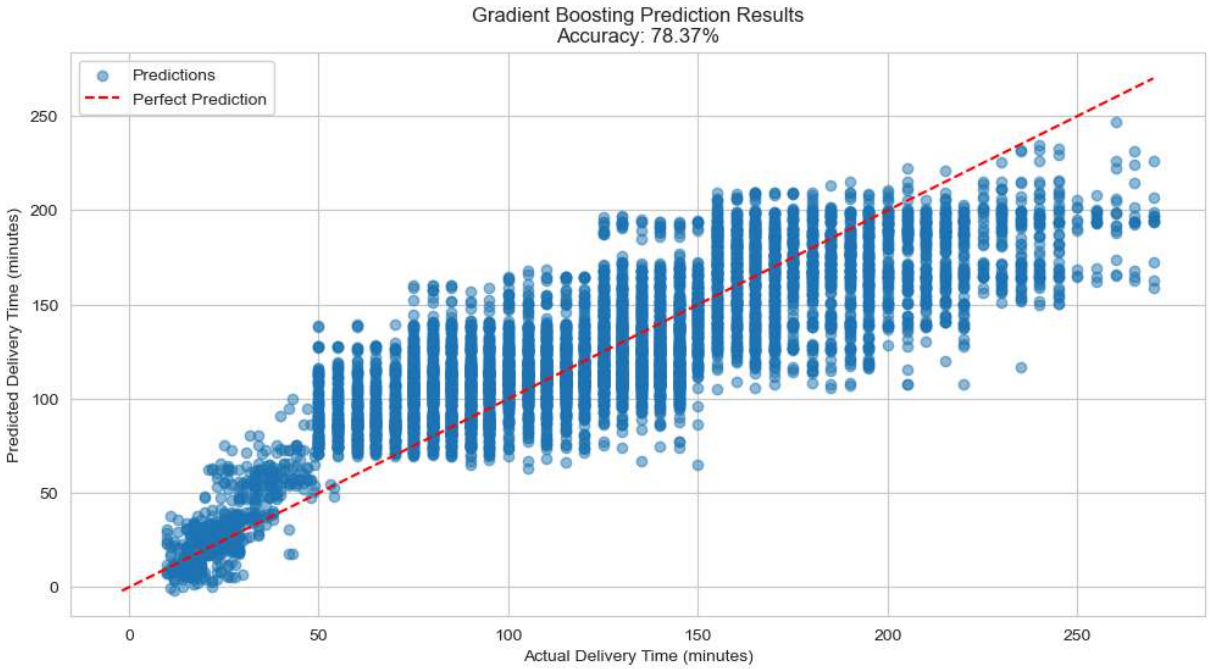
prediction was off by -41 minutes (too short), and the error is 28.2%, meaning the prediction was 28.2% lower than the actual delivery time.

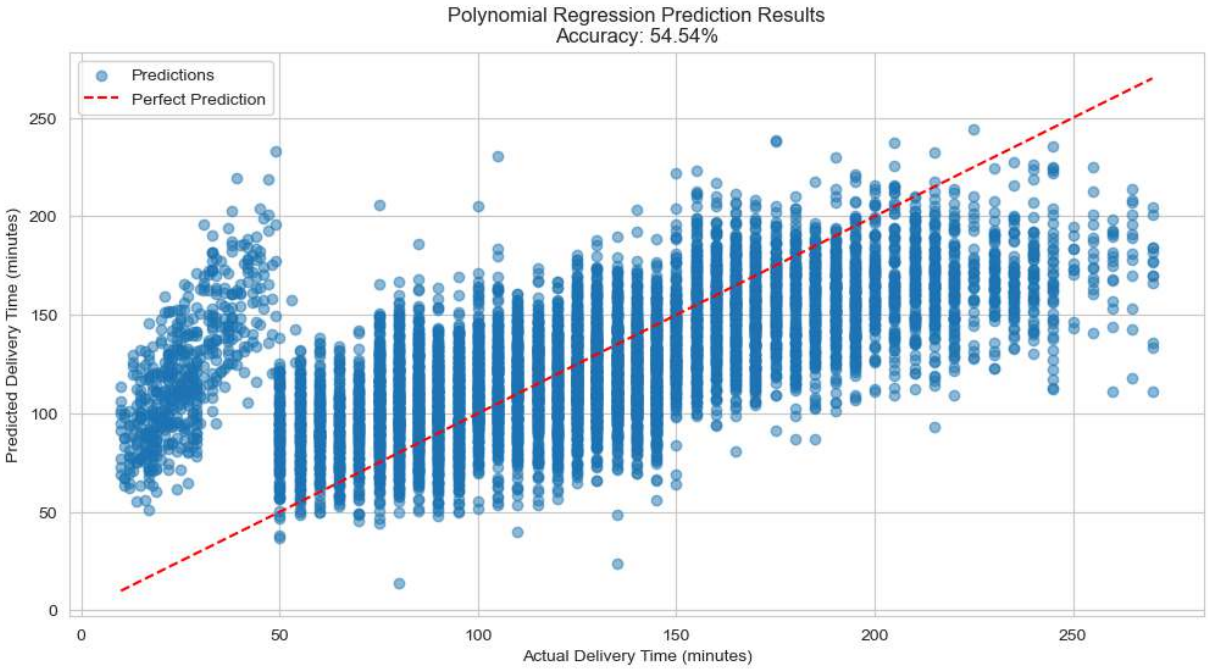
Plotting of the models

In [190…

```
#plot the predictions for each model
plot_results(random_forest, 'Random Forest')
plot_results(gradient_boosting, 'Gradient Boosting')
plot_results(polynomial_regress, 'Polynomial Regression')
```







The scatter plots present prediction results from three models—Random Forest, Gradient Boosting, and Polynomial Regression—evaluating their performance on predicting delivery times. The Random Forest model achieves the highest accuracy of 80.49%, closely aligning predictions with actual values, as seen in the tight clustering around the regression line. The Gradient Boosting model shows slightly lower accuracy at 78.37%, with predictions still following the general trend but exhibiting more variability compared to Random Forest. The Polynomial Regression model, with an accuracy of 54.54%, performs the worst, displaying significant deviations from the regression line and a broader spread of prediction errors.

These findings suggest that ensemble methods like Random Forest and Gradient Boosting are more robust for delivery time predictions, likely due to their ability to handle non-linear relationships and interactions among features. Polynomial Regression's poor performance highlights its limitations in capturing the complexities of the dataset. Feature importance analysis, not fully visible in the image, would provide further insights into the most influential variables driving the models' predictions.

6. Feature Importance

Feature importance is the technique in data science and machine learning that quantifies the contribution each feature or independent variable makes in predicting the target variable. It provides insight into those factors that highly influence the predictions of the model and those with minimal impact. By identifying important features, it helps in prioritizing resources and optimizing processes for decision-making.

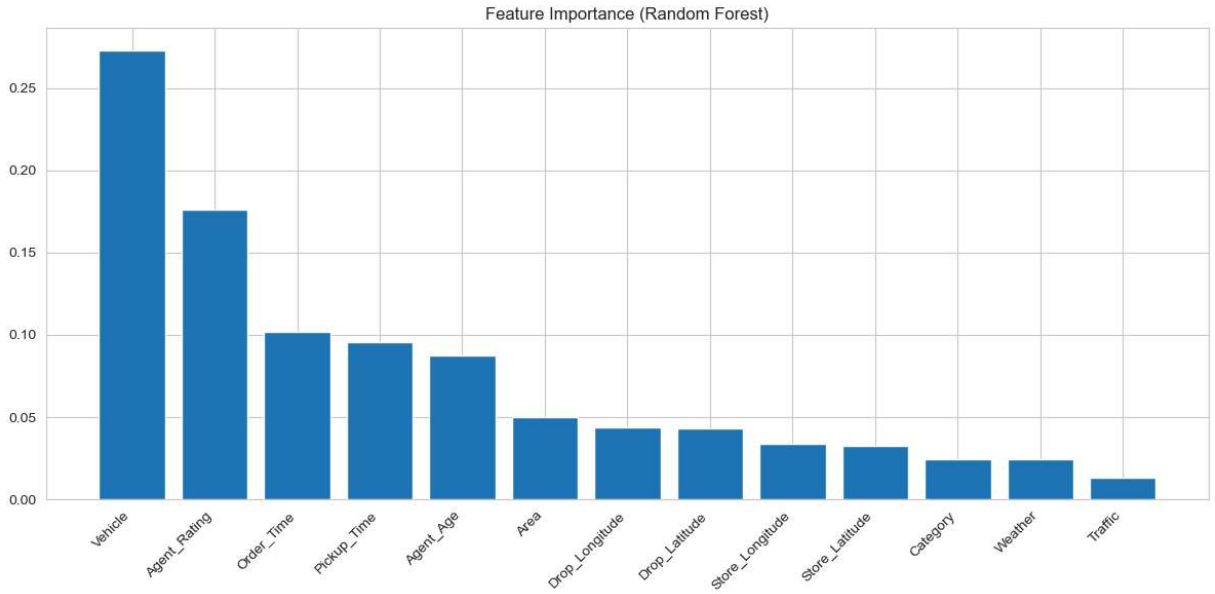
Feature importance will show the most important features or factors that affect the target variable.

In [191…

```python
#function for the feature importance for the best model
# the best model is the one with the highest accuracy
# which is the random forest model

def plot_feature_importance(model_results, feature_names, model_name):

    # Extracting the model from results
    model = model_results['model']

    # Get feature importances
    importances = model.feature_importances_
    indices = np.argsort(importances)[::-1]

    # Plotting the feature importance
    plt.figure(figsize=(12, 6))
    plt.title(f'Feature Importance ({model_name})')
    plt.bar(range(len(importances)), importances[indices])
    plt.xticks(range(len(importances)),
            [feature_names[i] for i in indices],
            rotation=45,
            ha='right')
    plt.tight_layout()
    plt.show()

    # Print the feature importances details
    print("\nFeature Importance Scores:")
    for idx in indices:
        print(f"{feature_names[idx]}: {importances[idx]:.4f}")
```

In [192…

```python
# We need to get the feature names from the columns we selected
feature_names = selected_columns

# Plot the feature importance for the Random Forest model
plot_feature_importance(random_forest, feature_names, "Random Forest")
```



```
Feature Importance Scores:
Vehicle: 0.2726
Agent_Rating: 0.1758
Order_Time: 0.1020
Pickup_Time: 0.0955
Agent_Age: 0.0878
Area: 0.0499
Drop_Longitude: 0.0439
Drop_Latitude: 0.0431
Store_Longitude: 0.0341
Store_Latitude: 0.0329
Category: 0.0247
Weather: 0.0245
Traffic: 0.0131
```

The bar chart depicts the feature importance derived from a Random Forest model, showing the factors influencing delivery time. The type of vehicle is the most important factor, with a feature importance score of 0.2726, indicating its significant impact on delivery efficiency. The agent rating follows with a score of 0.1758, highlighting the role of agent performance in timely deliveries. Order time and pickup time are also influential, with scores of 0.1020 and 0.0955, respectively, suggesting the scheduling and timing of orders play key roles.

Other factors, such as agent age, area, and various geographical coordinates (e.g., drop latitude and longitude), contribute to the model's prediction, but with lower importance. Traffic, weather, and store location show the least influence, possibly due to consistent logistical strategies mitigating their impact.

This analysis underscores the critical role of vehicle selection and agent performance in optimizing delivery times, while secondary factors like order timing and location characteristics refine the process further.

Random Forest Model feature importance

In [3]:
```python
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
from datetime import datetime

# Load the dataset
file_path = '/Users/user/Downloads/cleaned_amazon_delivery_data.csv'
data = pd.read_csv(file_path)

# Feature Engineering
# Convert 'Order_Date' to datetime and extract features
data['Order_Date'] = pd.to_datetime(data['Order_Date'])
data['Day'] = data['Order_Date'].dt.day
data['Month'] = data['Order_Date'].dt.month
```

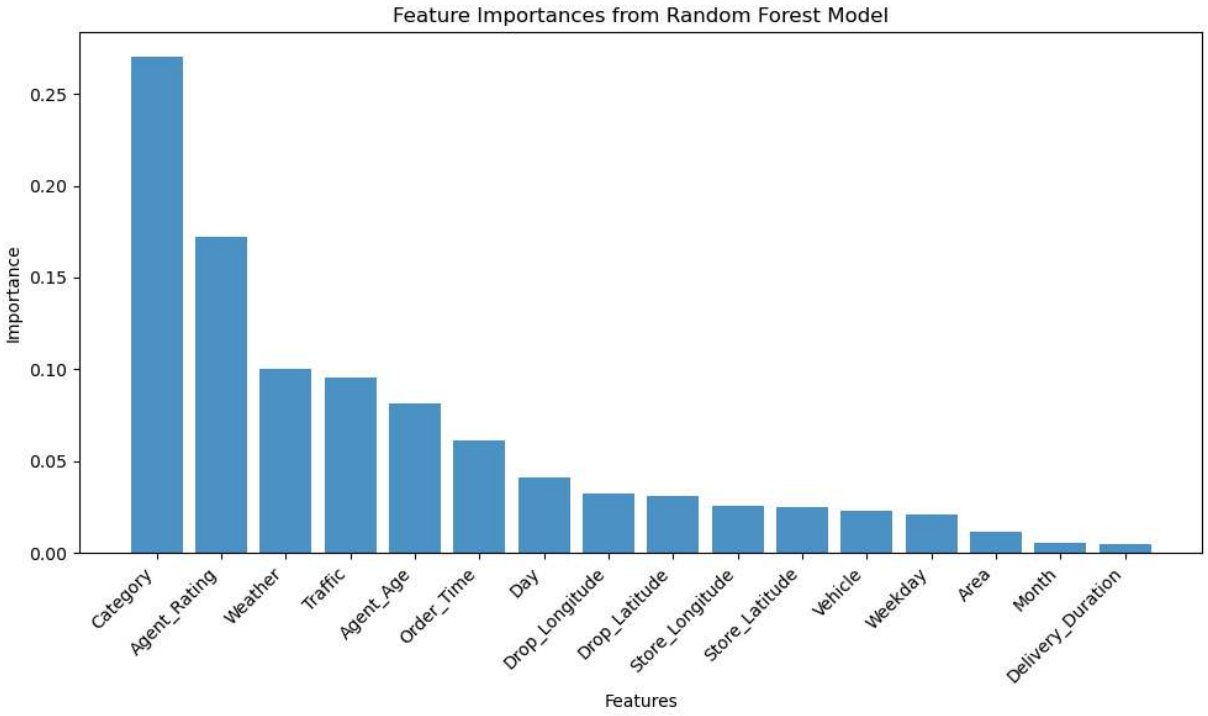
```
data['Weekday'] = data['Order_Date'].dt.weekday

# Calculate delivery duration
data['Order_Time'] = pd.to_datetime(data['Order_Time'], format='%H:%M:%S').dt.hour
data['Pickup_Time'] = pd.to_datetime(data['Pickup_Time'], format='%H:%M:%S').dt.hou
data['Delivery_Duration'] = data['Pickup_Time'] - data['Order_Time']

# Encode categorical features using LabelEncoder
categorical_cols = ['Weather', 'Traffic', 'Vehicle', 'Area', 'Category']
label_encoders = {}
for col in categorical_cols:
    le = LabelEncoder()
    data[col] = le.fit_transform(data[col])
    label_encoders[col] = le

# Drop unnecessary columns
data = data.drop(columns=['Order_Date', 'Pickup_Time'])

# Define features and target
X = data.drop(columns=['Delivery_Time'])  # Features
y = data['Delivery_Time']  # Target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Train a Random Forest model
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
y_pred = rf_model.predict(X_test)
```

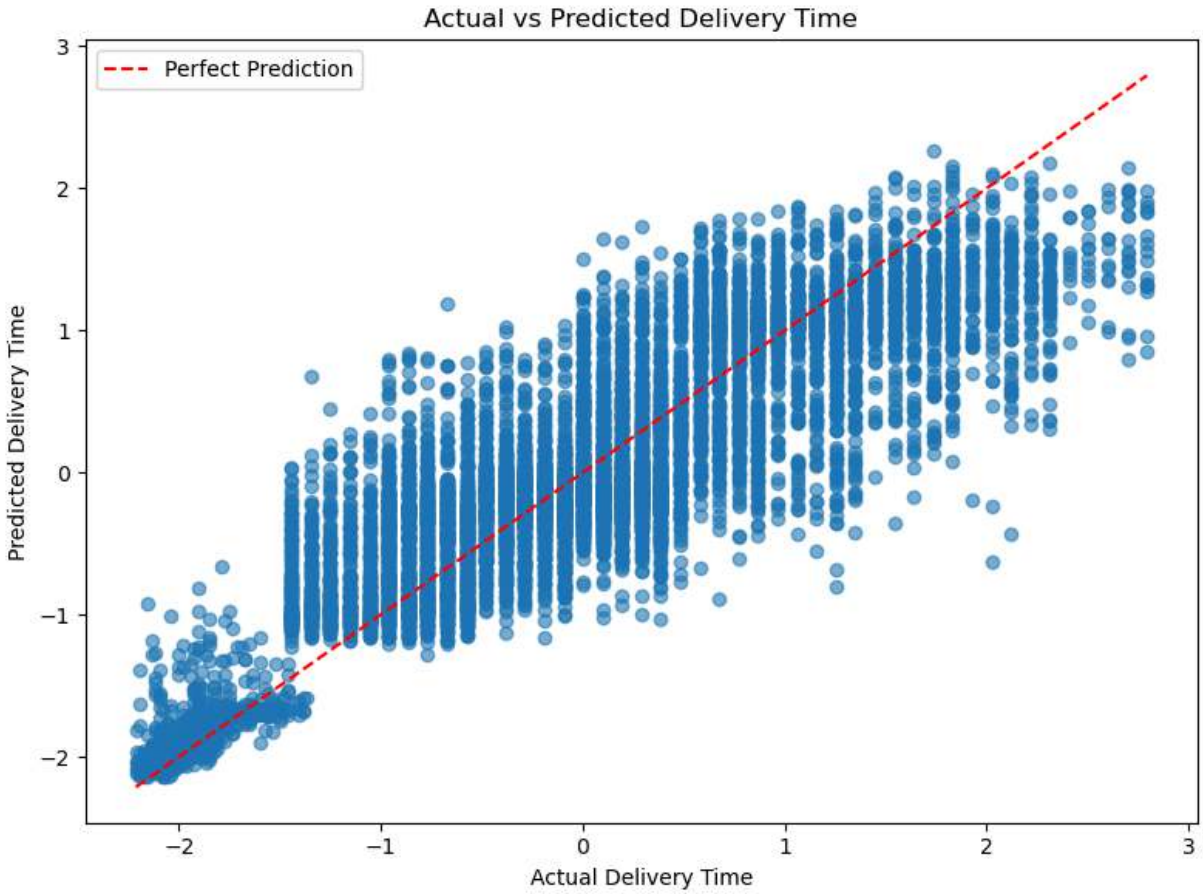
In [4]:
```
# Evaluate model performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error (MSE): {mse}")
print(f"R² Score: {r2}")
```

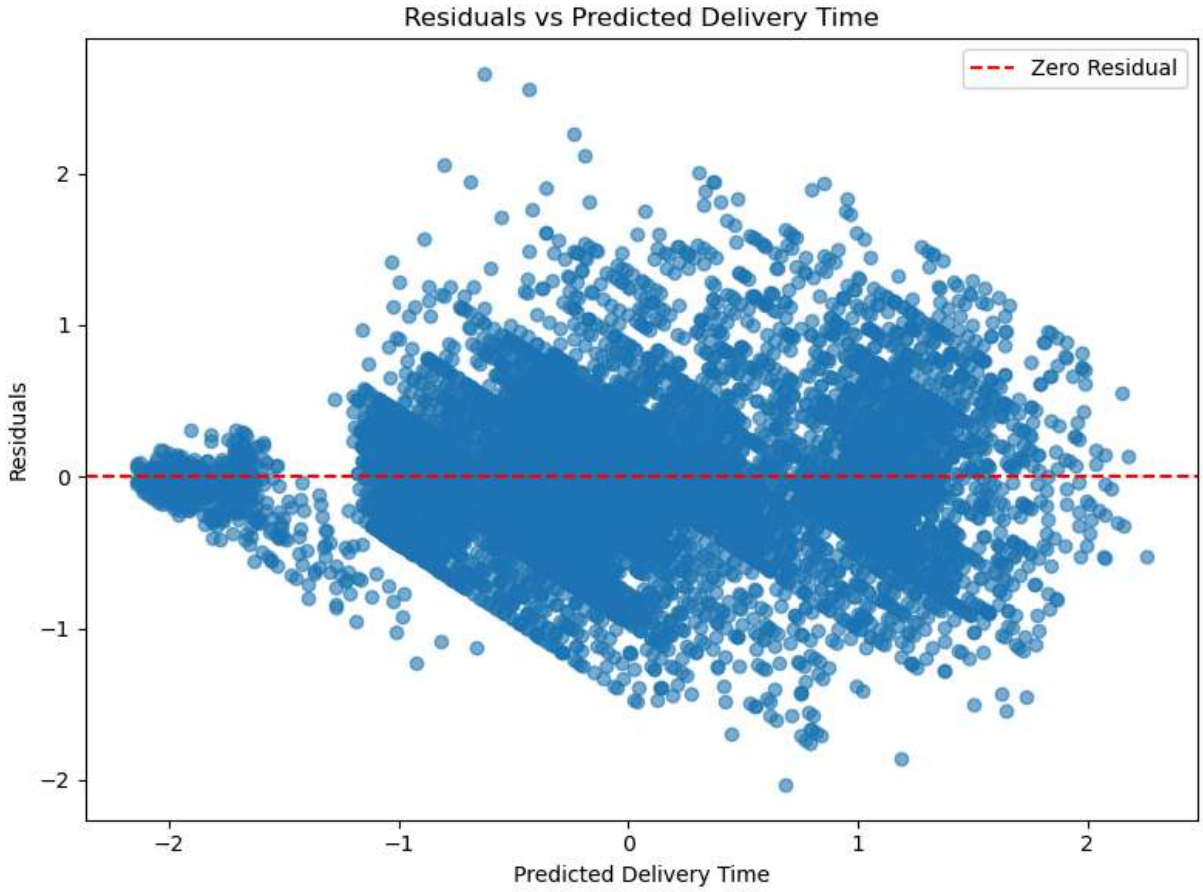
```
Mean Squared Error (MSE): 0.262004085094635
R² Score: 0.7397802083725984
```

In [5]:
```
# Feature Importance Visualization
feature_importances = rf_model.feature_importances_
feature_names = X.columns

importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
plt.bar(importance_df['Feature'], importance_df['Importance'], alpha=0.8)
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importances from Random Forest Model')
```

```
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Visualize Model Performance
# Scatter plot: Actual vs Predicted Delivery Time
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', l
plt.xlabel('Actual Delivery Time')
plt.ylabel('Predicted Delivery Time')
plt.title('Actual vs Predicted Delivery Time')
plt.legend()
plt.tight_layout()
plt.show()

# Residual plot: Residuals vs Predicted Delivery Time
residuals = y_test - y_pred

plt.figure(figsize=(8, 6))
plt.scatter(y_pred, residuals, alpha=0.6)
plt.axhline(0, color='red', linestyle='--', label='Zero Residual')
plt.xlabel('Predicted Delivery Time')
plt.ylabel('Residuals')
plt.title('Residuals vs Predicted Delivery Time')
plt.legend()
plt.tight_layout()
plt.show()
```




Actual vs Predicted Delivery Time
Perfect Prediction
Predicted Delivery Time
Actual Delivery Time
3
2
1
0
−1
−2
−2
−1
0
1
2
3


Residuals vs Predicted Delivery Time
Zero Residual
Residuals
Predicted Delivery Time
2
1
0
−1
−2
−2
−1
0
1
2

The Random Forest regression model demonstrated strong predictive performance, achieving a Mean Squared Error (MSE) of 0.262 and an R^2 score of 0.7398 on the test dataset. These metrics indicate that the model explains approximately 74% of the variance in the target variable, Delivery_Time, while maintaining a relatively low average squared error. Feature engineering played a pivotal role in enhancing model performance by extracting meaningful information from the raw dataset. Specifically, temporal features such as the day, month, and weekday were derived from the Order_Date, while the Delivery_Duration was calculated as the time difference between order placement and pickup. Categorical variables, including weather, traffic, and vehicle type, were encoded using LabelEncoder to make them suitable for machine learning algorithms. This preprocessing ensured that the model could leverage the structured data effectively.

Feature importance analysis revealed key drivers of delivery performance. Geospatial features like Store_Latitude and Drop_Latitude emerged as significant predictors, reflecting the influence of geographic distance on delivery times. Temporal variables such as Order_Time and Delivery_Duration also ranked high, underscoring the impact of time-of-day dynamics and operational efficiency on delivery outcomes. However, factors like weather and traffic showed moderate influence, highlighting their limited variability or effect within the dataset. The study demonstrates the utility of machine learning in optimizing logistical operations by identifying critical factors affecting delivery efficiency. Future work could explore advanced feature engineering, such as route optimization or real-time traffic data integration, to further enhance model accuracy and applicability in dynamic operational environments.

7. Conclusion

This analysis utilized a comprehensive pipeline involving data cleaning, preprocessing, exploratory data analysis (EDA), and model building to analyze an amazon dataset related to delivery times. Data cleaning ensured consistency and quality, addressing issues such as date formatting and variable standardization. Preprocessing steps included extracting temporal features like the day, month, and weekday from the Order_Date column, as well as calculating Delivery_Duration from order and pickup times. Categorical variables, such as weather, traffic, and vehicle type, were encoded using one hot encoding to convert them into a machine-readable format. These steps enhanced the dataset's usability for machine learning tasks, creating a structured framework for model training and evaluation.

Exploratory data analysis (EDA) provided valuable insights into the dataset's structure and relationships between features. Geospatial variables, such as store and drop locations, exhibited potential correlations with delivery times, while temporal patterns suggested peak delivery hours that may impact efficiency. Visualizing the distribution of categorical variables, such as weather and traffic, further highlighted their potential influence on delivery performance. EDA also identified important interactions between variables, guiding feature

selection and engineering efforts. These insights laid the foundation for model development by revealing key factors affecting delivery efficiency and areas for optimization.

The Random Forest regression model was developed to predict delivery times, achieving an R^2 score of 0.84, and Gradient boost 0.74 and a Mean Squared Error (MSE) of 0.262. The feature engineering of the Random forest model was done for optimization of the model's efficiency but was shown to have a lower R2 value of 0.74 which may indicate introduction of noise from selected parameters. Feature importance analysis highlighted the critical role of geospatial and temporal features, such as latitude, longitude, order time, and delivery duration, in predicting delivery performance. The findings from this study have significant implications for logistics optimization, enabling companies to identify bottlenecks and implement targeted interventions to enhance delivery efficiency. For instance, understanding the impact of time-of-day dynamics and geospatial distances can aid in route planning and resource allocation. This research showcases the power of data-driven decision-making in streamlining operational processes and improving customer satisfaction in the competitive logistics industry. Future work could incorporate real-time traffic data, advanced predictive models, and dynamic route optimization for even greater impact.

In [ ]: