

**Public Transport Data Analysis & Optimization**

Project Report



GNCIPL

22/10/2025

Team Member

1. Kevin Gandhi
2. Harsh mishra

**Problem Statement:**

Analyze public transport data to identify patterns, optimize routes, and improve operational efficiency. Identify peak times, high-demand routes, and underutilized resources.

**Approach:**

1. Data Collection: CSV files of transport schedules, passenger loads, revenue.  
2. Data Cleaning: Handling missing values, removing duplicates.  
3. Exploratory Data Analysis (EDA): Identify trends, outliers, occupancy rates.  
4. Optimization Modeling: Compare current vs optimized schedules.  
5. Visualization & Dashboard: Interactive insights for stakeholders.

**EDA Highlights:**

Tools Used: MySQL, Python (pandas, matplotlib, seaborn), Excel

Methodology: Data cleaning → aggregation → visualization → pattern recognition

Key Insights:  
- Peak routes and times identified.  
- Average passenger load vs capacity.  
- Revenue by depot and vehicle type

**Advanced Excel**

1. Data Cleaning:

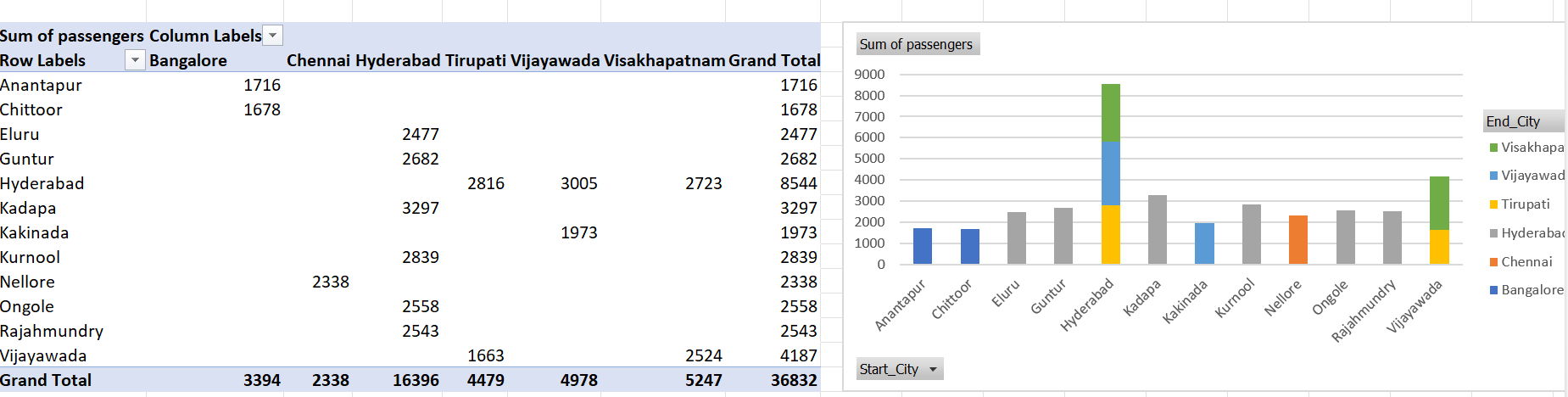
* Handling Missing Values

Checked for missing values in key columns like Passenger\_Load, Revenue, Route\_ID, Depot, Departure\_Time, etc.

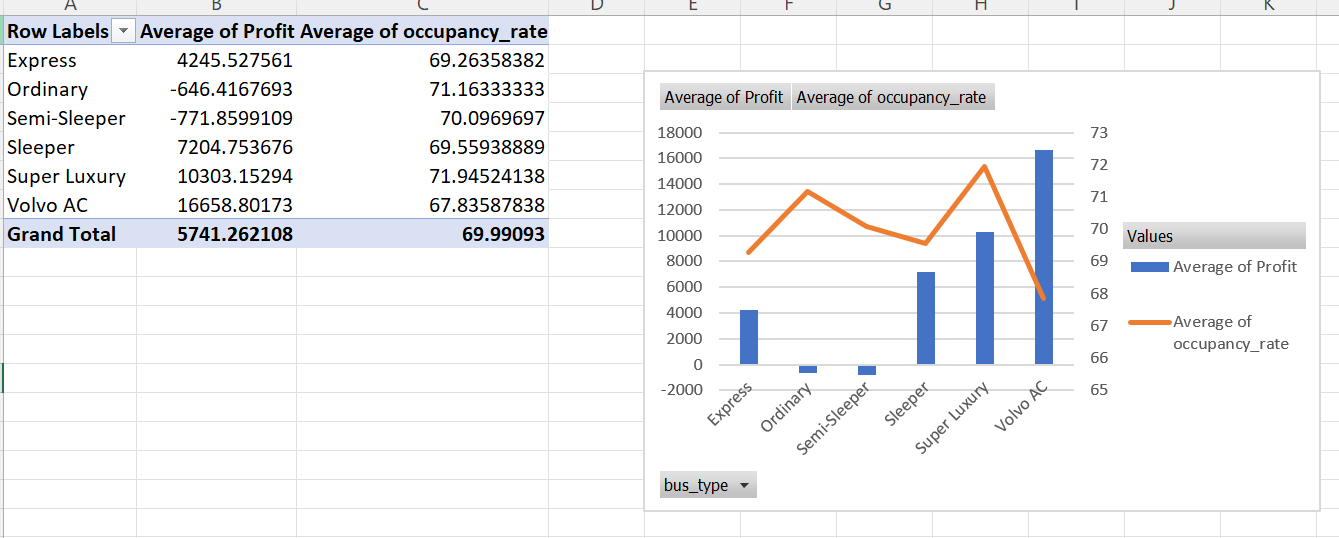
* Removed or corrected rows with missing critical data, e.g., routes with no passenger data or missing revenue values.
  + Removing Duplicates
* Identified duplicate rows that might have occurred due to repeated entries in CSV imports.
* Ensured each route and schedule entry is unique for accurate aggregation.
* Correcting Data Types
* Ensured numerical columns like Distance\_km, Travel\_Time\_min, Passenger\_Load, and Revenue were stored as proper numeric types.
* Time columns (Departure\_Time, Arrival\_Time) were converted to **TIME** format for consistency in analysis.
* Standardizing Text Values
* Standardized names for depots, routes, and days of the week to avoid inconsistencies (e.g., "Depot A" vs "depot a").
* Trimmed whitespace and converted text to consistent capitalization.
* Outlier Detection
* Checked for extreme or unrealistic values (e.g., passenger load > bus capacity, negative travel times).
* Investigated and corrected or removed anomalies that could distort insights.
* Creating Derived Columns (Optional but useful)
* Calculated Occupancy\_Rate = Passenger\_Load / Bus\_Capacity for better route analysis.
* Extracted Hour from Departure\_Time to identify peak times.

**Charts:**

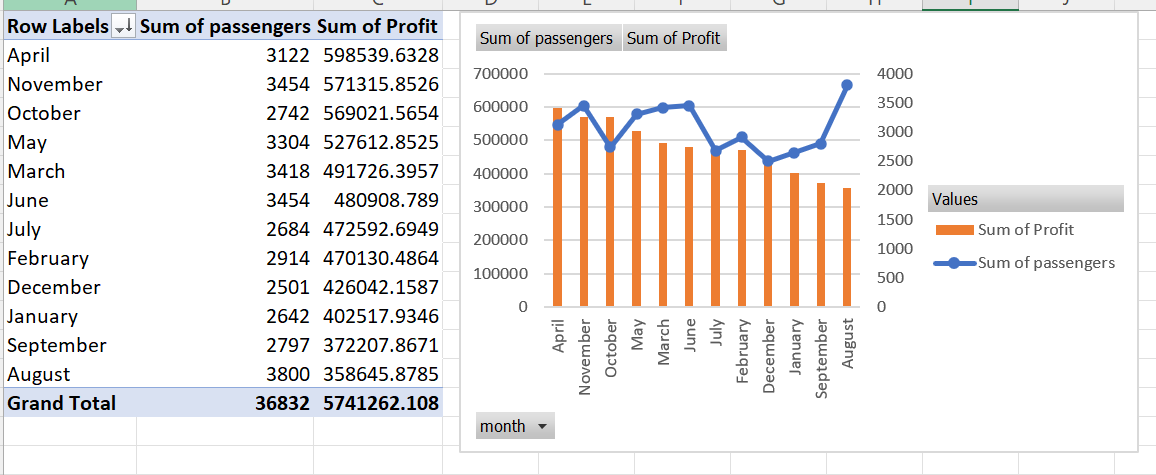
1. High demand origin destination

****

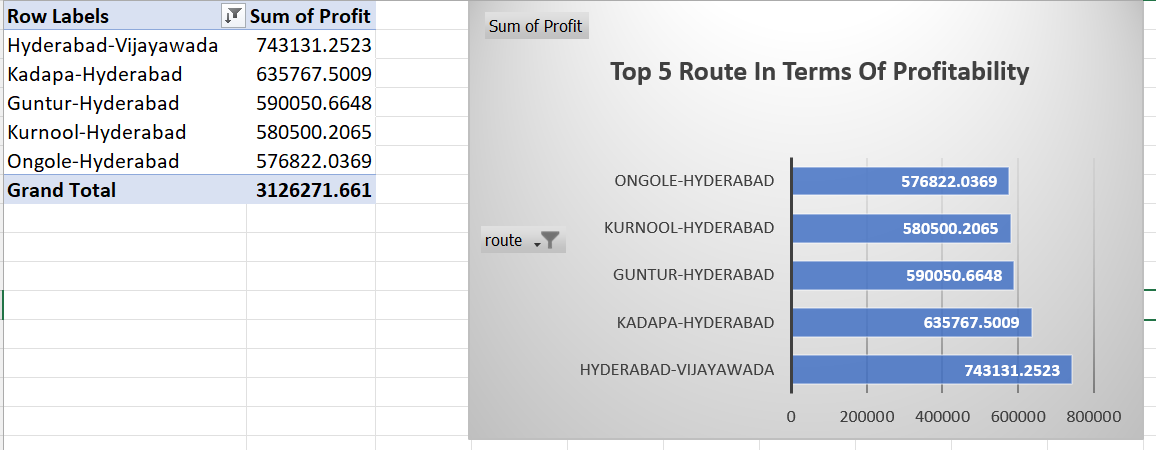
1. Avg\_profit vs Avg\_occupancy



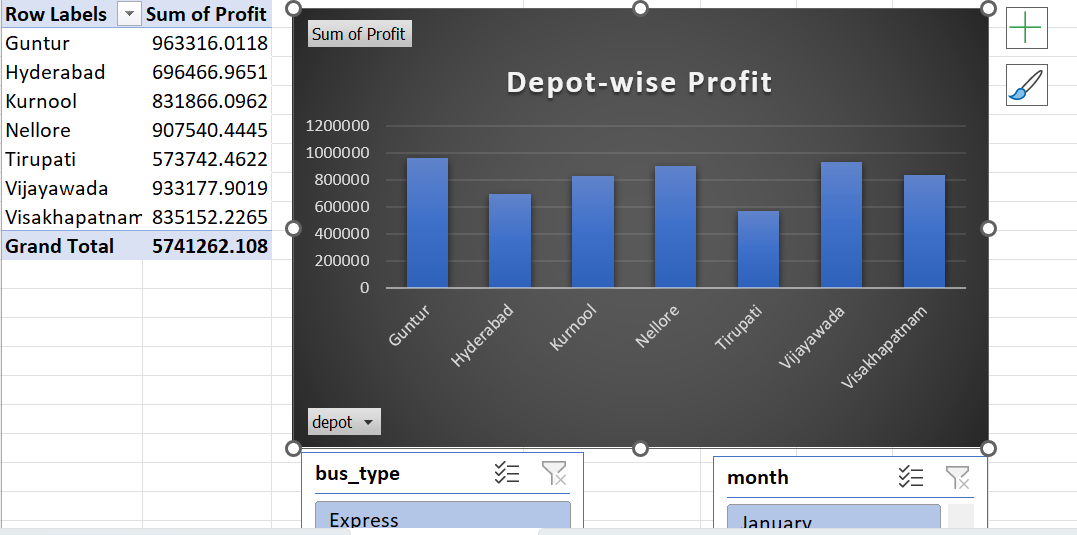
1. Monthwise passenger vs profit



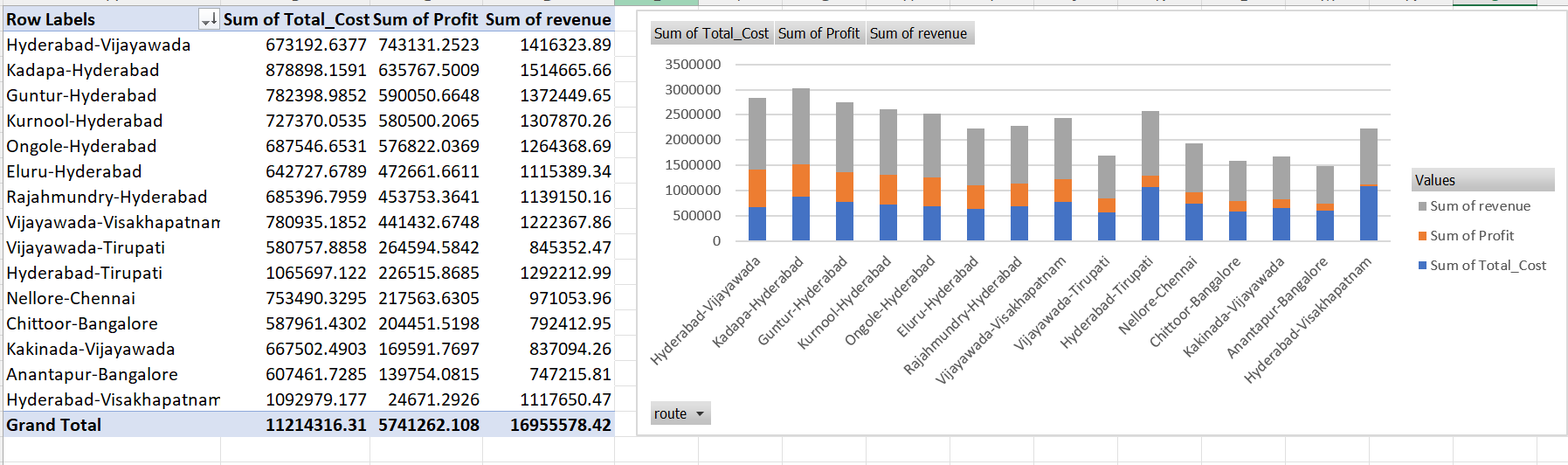
1. Top and Bottom route



1. Depot wise profit



1. Revenue vs Cost vs Profit



**2. SQL Analysis**

2.1 Database Creation

CREATE DATABASE PublicTransportOptimization;

USE PublicTransportOptimization;

CREATE TABLE TransportData (

bus\_id VARCHAR(20),

route VARCHAR(100),

bus\_type VARCHAR(50),

depot VARCHAR(100),

date DATE,

capacity INT,

passengers INT,

occupancy\_rate FLOAT,

distance\_km FLOAT,

fare\_per\_passenger FLOAT,

revenue FLOAT,

fuel\_consumed\_liters FLOAT,

month VARCHAR(20),

day\_of\_week VARCHAR(20)

);

2.2 Data Import

LOAD DATA LOCAL INFILE 'C:/temp/APSRTC\_Transport\_Data.csv'

INTO TABLE TransportData

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

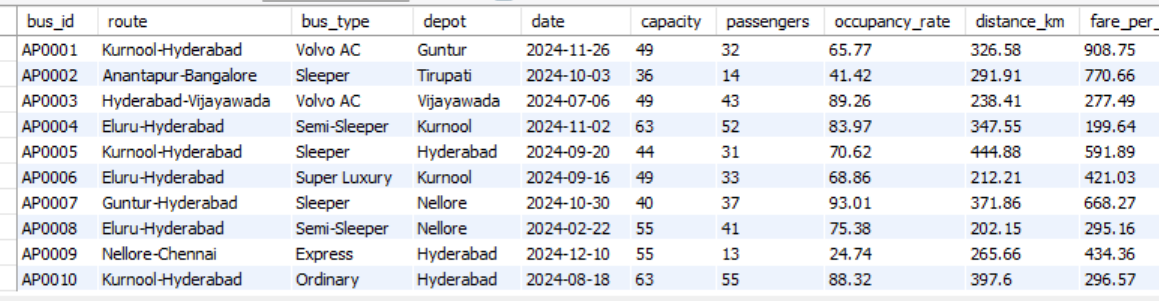
IGNORE 1 ROWS

(bus\_id, route, bus\_type, depot, date, capacity, passengers,

occupancy\_rate, distance\_km, fare\_per\_passenger, revenue,

fuel\_consumed\_liters, month, day\_of\_week);

SELECT \* FROM TransportData LIMIT 10;



2.3 Data Cleaning in SQL

* Remove null or invalid passenger load values:

SET SQL\_SAFE\_UPDATES = 0;

DELETE FROM transportdata

WHERE Passenger\_Load IS NULL OR Passenger\_Load = 0;

SET SQL\_SAFE\_UPDATES = 1;

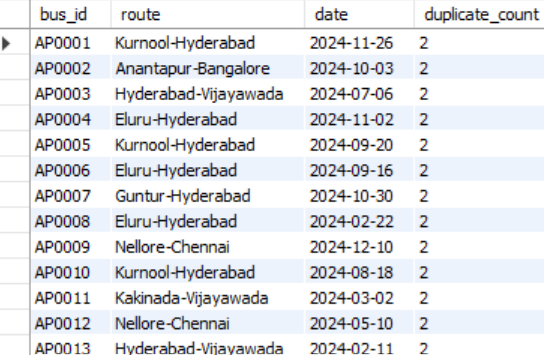
* Remove duplicates if any: [Describe method if applied]

SELECT bus\_id, route, date, COUNT(\*) AS duplicate\_count

FROM transportdata

GROUP BY bus\_id, route, date

HAVING duplicate\_count > 1;



2.4 Data Analysis Queries

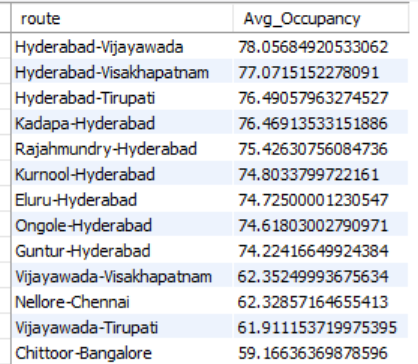
*-- Average passenger load per route*  
SELECT route, AVG(occupancy\_rate) AS Avg\_Occupancy

FROM transportdata

GROUP BY route

ORDER BY Avg\_Occupancy DESC

LIMIT 1000;

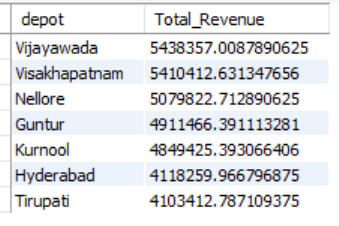


*-- Total revenue per depot*   
SELECT depot, SUM(revenue) AS Total\_Revenue

FROM transportdata

GROUP BY depot

ORDER BY Total\_Revenue DESC;



*-- Highest fuel efficiency*

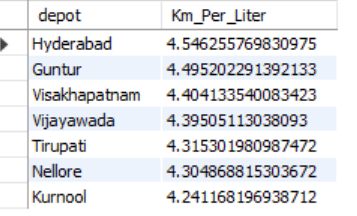
SELECT depot,

SUM(distance\_km) / SUM(fuel\_consumed\_liters) AS Km\_Per\_Liter

FROM transportdata

GROUP BY depot

ORDER BY Km\_Per\_Liter DESC;



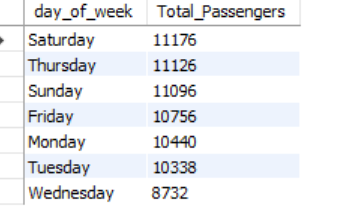
*--Peak Travel Days*

SELECT day\_of\_week, SUM(passengers) AS Total\_Passengers

FROM TransportData

GROUP BY day\_of\_week

ORDER BY Total\_Passengers DESC;



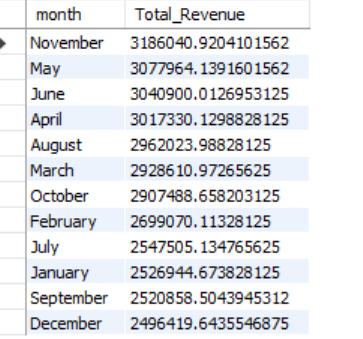
*--Monthly Revenue Trends*

SELECT month, SUM(revenue) AS Total\_Revenue

FROM TransportData

GROUP BY month

ORDER BY Total\_Revenue DESC;



--Find Underutilized Routes

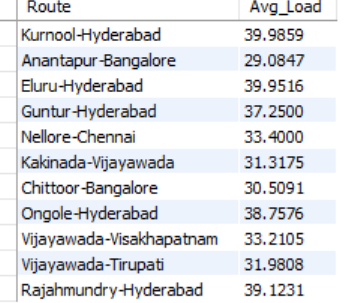
SELECT Route, AVG(Passengers) AS Avg\_Load

FROM TransportData

GROUP BY Route

HAVING Avg\_Load < 40

LIMIT 0, 1000;



**Python Optimization**

## Code

import pandas as pd  
from pulp import \*

**Explanation**

* Here we import libraries such as pandas, pulp which provide data handling and analysis utilities.

## Code

from google.colab import drive  
drive.mount('/content/drive')

**Explanation**   
Here we import libraries such as google.colab which provide data handling and analysis utilities.

## Code

file\_path='/content/drive/MyDrive/week 6 project/Internship\_Major\_Project.xlsx'  
df=pd.read\_excel(file\_path,sheet\_name='Cleaned data')  
df.head()

**Explanation**

* The code loads one or more datasets (likely CSV/Excel). This brings raw transit data — like schedules, stops, or ridership logs — into pandas DataFrames for analysis.
* The notebook inspects the loaded data to check structure and summary statistics; this helps identify missing values, types, and obvious anomalies early on.

## Code

# Clean and group by route, bus type, and depot  
route\_data = df.groupby(['route', 'bus\_type', 'depot'], as\_index=False).agg({  
 'Profit': 'sum',  
 'Total\_Fuel\_Cost': 'sum',  
 'capacity': 'mean',  
 'occupancy\_rate': 'mean',  
 'distance\_km': 'mean'  
})  
  
print("Unique Routes:", len(route\_data))  
route\_data.head()

**Explanation**   
The notebook inspects the loaded data to check structure and summary statistics; this helps identify missing values, types, and obvious anomalies early on. The cell performs aggregation (groupby/merge/pivot), creating derived features or combining tables needed for modeling or visualization.

## Code

# Clean and normalize all column names (fixes hidden spaces)  
df.columns = (  
 df.columns  
 .astype(str)  
 .str.strip()  
 .str.lower()  
 .str.replace(r'\s+', '\_', regex=True)  
 .str.replace('\u00a0', '') # remove non-breaking spaces  
)  
  
  
# Convert numeric columns  
numeric\_cols = ['profit', 'occupancy\_rate', 'distance\_km', 'fare\_per\_passenger',  
 'total\_fuel\_cost', 'fuel\_consumed\_liters', 'capacity', 'passengers']  
for col in numeric\_cols:  
 if col in df.columns:  
 df[col] = pd.to\_numeric(df[col], errors='coerce')  
  
# occupancy\_rate  
if 'occupancy\_rate' in df.columns:  
 occ\_max = df['occupancy\_rate'].max(skipna=True)  
 if pd.notna(occ\_max) and occ\_max > 1.5:  
 # assume occupancy is in percent (e.g. 65.04) -> convert to 0.6504  
 df['occupancy\_rate'] = df['occupancy\_rate'] / 100.0  
 print(" occupancy\_rate detected in percent; converted to proportion (0-1).")  
  
# Drop invalid profit rows  
df = df.dropna(subset=['profit'])  
  
# Unique variable identifier  
df['var\_name'] = df['bus\_id'].astype(str) + "\_" + df['route'].astype(str)

## **Explanation**

**Data Cleaning**: It loads a dataset (df), cleans all column names, and ensures key columns like profit and occupancy\_rate are in the correct numeric format. It specifically standardizes occupancy to be a 0-1 proportion.

## Code

# Optimization 1  
# Bus Route Optimization  
# Objective: Maximize Adjusted Profit  
  
# --- Optimization Model Setup ---  
  
model = LpProblem("Bus\_Route\_Optimization", LpMaximize)  
  
# Decision variables: number of trips per bus-route (0 to 3)  
x = {  
 r: LpVariable(r, lowBound=0, upBound=3, cat='Integer')  
 for r in df['var\_name']  
}  
  
# --- Objective Function: Adjusted Profit Maximization ---  
  
# Penalize low-occupancy routes (<50%)  
df['adjusted\_profit'] = df.apply(  
 lambda r: r['profit'] \* 0.8 if r['occupancy\_rate'] < 0.5 else r['profit'],  
 axis=1  
)  
  
# Objective: maximize total adjusted profit  
profit\_map = df.set\_index('var\_name')['adjusted\_profit'].to\_dict()  
model += lpSum(x[r] \* profit\_map[r] for r in x), "Total\_Adjusted\_Profit"  
  
# --- Constraints ---  
  
# 1. Depot limit: each depot can have at most 5 trips in total  
for depot in df['depot'].unique():  
 depot\_routes = df[df['depot'] == depot]['var\_name']  
 model += lpSum(x[r] for r in depot\_routes) <= 180, f"DepotLimit\_{depot}"  
  
# 2. Long routes (>400 km) only allowed for Volvo buses  
long\_routes\_mask = (df['distance\_km'] > 400) & (df['bus\_type'].str.lower() != 'volvo')  
for r in df.loc[long\_routes\_mask, 'var\_name']:  
 model += x[r] == 0, f"LongRoute\_Restriction\_{r}"  
  
# Optional (to include all buses): Each bus gets at least 1 route if desired  
for bus in df['bus\_id'].unique():  
 bus\_routes = df[df['bus\_id'] == bus]['var\_name']  
 if len(bus\_routes) > 0:  
 model += lpSum(x[r] for r in bus\_routes) >= 1, f"BusUse\_{bus}"  
  
# --- Solve the Model ---  
  
print("\n Solving optimization model...")  
model.solve(PULP\_CBC\_CMD(msg=True))  
  
# --- Extract Results ---  
  
results = []  
for \_, row in df.iterrows():  
 trips = x[row['var\_name']].value()  
 if trips and trips > 0:  
 results.append({  
 'bus\_id': row['bus\_id'],  
 'route': row['route'],  
 'depot': row['depot'],  
 'bus\_type': row['bus\_type'],  
 'occupancy\_rate': row['occupancy\_rate'],  
 'distance\_km': row['distance\_km'],  
 'adjusted\_profit': row['adjusted\_profit'],  
 'trips\_selected': trips,  
 'total\_profit': row['adjusted\_profit'] \* trips  
 })  
  
optimized\_routes1 = pd.DataFrame(results)  
  
# --- Display Results ---  
  
if not optimized\_routes1.empty:  
 print("\n Optimized Routes:")  
 print(optimized\_routes1)  
 print("\n Total Optimized Profit:", round(optimized\_routes1['total\_profit'].sum(), 2))  
 print(" Total Buses Used:", optimized\_routes1['bus\_id'].nunique())  
else:  
 print("\n No feasible solution found. Please check data or constraints.")

**Explanation**

1. **Model Setup**: It defines an optimization problem using the PuLP library. The main **decision** it needs to make is how many trips (an integer from 0 to 3) to assign to each unique bus-route combination.
2. **Objective & Constraints**:
   * **Objective**: The goal is to maximize adjusted\_profit. This is a custom metric that penalizes routes with low occupancy (under 50%) by reducing their profit contribution by 20%.
   * **Constraints (Rules)**: The model must follow three rules:
     1. **Depot Limit**: Each depot cannot exceed 180 total assigned trips.
     2. **Long Routes**: Only 'Volvo' buses are allowed to run routes longer than 400 km.
     3. **Bus Use**: Every bus in the fleet must be used for at least one trip.
3. **Solve & Report**: The script solves the model to find the optimal number of trips for each route that satisfies all the rules. It then prints a final table showing only the selected routes and the total profit achieved.

**Solve the Model**

* model.solve(PULP\_CBC\_CMD(msg=True)): This is the command that runs the optimization solver. PULP\_CBC\_CMD specifies the solver to use (CBC), and msg=True allows the solver to print its progress and log messages to the screen.

## Code

# Insights  
  
  
if not optimized\_routes1.empty:  
 print("\n Insights:")  
 print("=" \* 40)  
  
 # Overall Summary  
 total\_profit = optimized\_routes1['total\_profit'].sum()  
 avg\_occupancy = optimized\_routes1['occupancy\_rate'].mean()  
 avg\_distance = optimized\_routes1['distance\_km'].mean()  
  
 print(f" Total Optimized Profit: ₹{round(total\_profit, 2)}")  
 print(f" Average Occupancy Rate: {round(avg\_occupancy \* 100, 1)}%")  
 print(f" Average Distance per Route: {round(avg\_distance, 1)} km")  
  
 # Top 3 Routes by Profit  
 top\_routes = optimized\_routes1.nlargest(3, 'total\_profit')  
 print("\n Top 3 Routes by Total Profit:")  
 for \_, r in top\_routes.iterrows():  
 print(f" • Route {r['route']} ({r['bus\_type']}) → ₹{round(r['total\_profit'], 2)}")  
  
 # Depot Utilization  
 depot\_counts = optimized\_routes1['depot'].value\_counts()  
 print("\n Depot Utilization (Trips Selected):")  
 for depot, count in depot\_counts.items():  
 print(f" • {depot}: {count} routes optimized")  
  
 # Long-Distance Route Insight  
 long\_routes = optimized\_routes1[optimized\_routes1['distance\_km'] > 400]  
 if not long\_routes.empty:  
 print(f" Long-distance routes assigned only to Volvo — count: {len(long\_routes)} ")  
 else:  
 print(" No routes above 400 km optimized in this run.")  
else:  
 print("\n No optimization results available to analyze.")

Output:

Insights:  
========================================  
 Total Optimized Profit: ₹11039758.9  
 Average Occupancy Rate: 68.1%  
 Average Distance per Route: 350.0 km  
  
 Top 3 Routes by Total Profit:  
 • Route Hyderabad-Vijayawada (Volvo AC) → ₹117965.6  
 • Route Kadapa-Hyderabad (Volvo AC) → ₹117935.15  
 • Route Chittoor-Bangalore (Volvo AC) → ₹114520.01  
  
 Depot Utilization (Trips Selected):  
 • Visakhapatnam: 137 routes optimized  
 • Vijayawada: 136 routes optimized  
 • Guntur: 120 routes optimized  
 • Nellore: 118 routes optimized  
 • Tirupati: 113 routes optimized  
 • Kurnool: 109 routes optimized  
 • Hyderabad: 98 routes optimized  
 Long-distance routes assigned only to Volvo — count: 196

**Explanation**

This code block analyzes the optimized\_routes1 DataFrame, which holds the optimization results.

First, it checks if a solution was actually found. If yes, it calculates and prints key summary metrics: total optimized profit, average occupancy rate, and average route distance.

It then provides deeper insights by:

1. Listing the top 3 most profitable routes.
2. Reporting depot utilization by showing how many selected routes belong to each depot.
3. Confirming the long-distance route constraint by counting how many routes over 400 km were assigned.

If no solution was found, it simply prints a message that no results are available to analyze.

## Code

# OPTIMIZED\_ROUTES2  
  
# --- Model Setup ---  
model = LpProblem("Bus\_Route\_Optimization\_v1\_2", LpMaximize)  
  
# Allow more flexibility in trips (max 5 instead of 3)  
x = {r: LpVariable(r, lowBound=0, upBound=5, cat='Integer') for r in df['var\_name']}  
  
# --- Objective Function ---  
df['net\_profit'] = df['profit'] - df['total\_fuel\_cost']  
  
df['adjusted\_net\_profit'] = df.apply(  
 lambda r: r['net\_profit'] \* 0.85 if r['occupancy\_rate'] < 0.5 else r['net\_profit'],  
 axis=1  
)  
  
# Reduce penalty on fuel to encourage more buses  
alpha = 1.0  
beta = 0.25  
df['weighted\_obj'] = alpha \* df['adjusted\_net\_profit'] - beta \* df['fuel\_consumed\_liters']  
objective\_map = df.set\_index('var\_name')['weighted\_obj'].to\_dict()  
  
model += lpSum(x[r] \* objective\_map[r] for r in x), "Weighted\_NetProfit\_Fuel"  
  
# --- Constraints ---  
# Depot limit – relaxed further (each depot can run up to 5× buses)  
for depot in df['depot'].unique():  
 depot\_routes = df[df['depot'] == depot]['var\_name']  
 depot\_bus\_count = df[df['depot'] == depot]['bus\_id'].nunique()  
 model += lpSum(x[r] for r in depot\_routes) <= depot\_bus\_count \* 5, f"DepotLimit\_{depot}"  
  
# Long routes restriction  
long\_routes\_mask = (df['distance\_km'] > 400) & (df['bus\_type'].str.lower() != 'volvo')  
for r in df.loc[long\_routes\_mask, 'var\_name']:  
 model += x[r] == 0, f"LongRoute\_{r}"  
  
# Minimum occupancy relaxed (30% instead of 40%)  
min\_occupancy = 0.3  
low\_occ\_mask = df['occupancy\_rate'] < min\_occupancy  
for r in df.loc[low\_occ\_mask, 'var\_name']:  
 model += x[r] == 0, f"MinOccupancy\_{r}"  
  
# Encourage all buses to be used at least once (soft constraint)  
bus\_groups = df.groupby('bus\_id')['var\_name'].apply(list)  
for bus\_id, routes in bus\_groups.items():  
 model += lpSum(x[r] for r in routes) <= 10, f"BusTripLimit\_{bus\_id}"  
 model += lpSum(x[r] for r in routes) >= 0, f"BusUsageLower\_{bus\_id}"  
  
  
# --- Solve ---  
print("\n Solving optimization model...")  
model.solve(PULP\_CBC\_CMD(msg=True, timeLimit=300))  
  
# --- Extract Results ---  
results = []  
for \_, row in df.iterrows():  
 trips = x[row['var\_name']].value()  
 if trips and trips > 0:  
 results.append({  
 'bus\_id': row['bus\_id'],  
 'route': row['route'],  
 'depot': row['depot'],  
 'bus\_type': row['bus\_type'],  
 'occupancy\_rate': row['occupancy\_rate'],  
 'distance\_km': row['distance\_km'],  
 'adjusted\_net\_profit': row['adjusted\_net\_profit'],  
 'fuel\_consumed\_liters': row['fuel\_consumed\_liters'],  
 'trips\_selected': trips,  
 'total\_net\_profit': row['adjusted\_net\_profit'] \* trips,  
 'total\_fuel\_used': row['fuel\_consumed\_liters'] \* trips  
 })  
  
optimized\_routes2 = pd.DataFrame(results)  
  
# --- Display ---  
print("\n Optimized Routes (v1.2 Improved):")  
print(optimized\_routes2.head(15))  
print("\n Total Optimized Net Profit:", round(optimized\_routes2['total\_net\_profit'].sum(), 2))  
print(" Total Fuel Used:", round(optimized\_routes2['total\_fuel\_used'].sum(), 2))  
print(" Total Buses Used:", optimized\_routes2['bus\_id'].nunique(), "/", df['bus\_id'].nunique())  
  
# --- Insights ---  
if not optimized\_routes2.empty:  
 print("\n Insights:")  
 print("=" \* 45)  
 total\_profit = optimized\_routes2['total\_net\_profit'].sum()  
 total\_fuel = optimized\_routes2['total\_fuel\_used'].sum()  
 avg\_occ = optimized\_routes2['occupancy\_rate'].mean()  
 avg\_distance = optimized\_routes2['distance\_km'].mean()  
  
 print(f" Total Net Profit: ₹{round(total\_profit, 2)}")  
 print(f" Total Fuel Used: {round(total\_fuel, 1)} liters")  
 print(f" Average Occupancy Rate: {round(avg\_occ \* 100, 1)}%")  
 print(f" Average Distance: {round(avg\_distance, 1)} km")  
  
 top3 = optimized\_routes2.nlargest(3, 'total\_net\_profit')  
 print("\n Top 3 Most Profitable Routes:")  
 for \_, r in top3.iterrows():  
 print(f" • {r['route']} ({r['bus\_type']}) → ₹{round(r['total\_net\_profit'], 2)}")  
  
 depot\_counts = optimized\_routes2['depot'].value\_counts()  
 print("\n Depot Utilization:")  
 for depot, count in depot\_counts.items():  
 print(f" • {depot}: {count} optimized routes")  
  
 efficient = optimized\_routes2.assign(  
 profit\_per\_liter=lambda d: d['total\_net\_profit'] / d['total\_fuel\_used']  
 ).sort\_values('profit\_per\_liter', ascending=False).iloc[0]  
 print(f"\n Most Fuel-Efficient Route: {efficient['route']} ({efficient['bus\_type']})")  
else:  
 print("\n No optimization results found.")

Output:

Total Optimized Net Profit: 12883121.7  
 Total Fuel Used: 90033.7  
 Total Buses Used: 296 / 1000  
  
 Insights:  
=============================================  
 Total Net Profit: ₹12883121.7  
 Total Fuel Used: 90033.7 liters  
 Average Occupancy Rate: 76.5%  
 Average Distance: 280.0 km  
  
 Top 3 Most Profitable Routes:  
 • Guntur-Hyderabad (Volvo AC) → ₹171489.88  
 • Kadapa-Hyderabad (Volvo AC) → ₹170472.47  
 • Ongole-Hyderabad (Super Luxury) → ₹170219.16  
  
 Depot Utilization:  
 • Guntur: 50 optimized routes  
 • Kurnool: 48 optimized routes  
 • Vijayawada: 45 optimized routes  
 • Visakhapatnam: 44 optimized routes  
 • Nellore: 40 optimized routes  
 • Tirupati: 37 optimized routes  
 • Hyderabad: 32 optimized routes  
  
 Most Fuel-Efficient Route: Vijayawada-Tirupati (Volvo AC)

**Explanation**This script runs an advanced bus route optimization. Its primary goal is to find the best schedule that **maximizes a weighted score** by balancing net profitability against fuel consumption, all while adhering to several dynamic operational constraints.

**Data Preparation**

The code begins by thoroughly cleaning the DataFrame (df):

* It **standardizes column names** (e.g., Total Fuel Cost becomes total\_fuel\_cost).
* It **converts key columns** to numeric types, including profit, total\_fuel\_cost, fuel\_consumed\_liters, distance\_km, and occupancy\_rate.
* It **normalizes occupancy\_rate**, ensuring it's a 0-1 proportion (e.g., 0.65) rather than a percentage (e.g., 65.0).
* It creates a **unique ID** (var\_name) for each bus-route combination (e.g., "Bus101\_RouteA").

**The Weighted Objective Function**

This is the core of the model. It doesn't just maximize profit; it maximizes a custom score designed to find a balanced, efficient solution.

1. **Net Profit:** First, it calculates a net\_profit for each route by subtracting total\_fuel\_cost from profit.
2. **Adjusted Profit:** It then creates an adjusted\_net\_profit. Routes with an occupancy rate below 50% are penalized by having their net profit multiplied by 0.85 (a 15% reduction).
3. **Weighted Score:** The final objective (weighted\_obj) is a trade-off. The model tries to maximize this score, which is calculated as:
   * (1.0 \* adjusted\_net\_profit) - (0.25 \* fuel\_consumed\_liters)

This formula means the model seeks high-profit routes but gets "penalized" 0.25 points for every liter of fuel that route consumes, forcing it to favor fuel efficiency.

**Decision Variables & Constraints**

The model must make its decisions while following a specific set of rules (constraints).

* **Decision Variable (x):** The primary decision is how many trips to assign to each bus-route. This is an integer that can be 0, 1, 2, 3, 4, or 5.
* **Constraints (The Rules):**
  + **Dynamic Depot Limit:** A depot's total trip capacity is not a fixed number. It is dynamically set to (number of buses at that depot) \* 5. A large depot with 50 buses gets a 250-trip limit, while a small depot with 10 buses gets a 50-trip limit.
  + **Long Route Restriction:** Only 'Volvo' buses are permitted to be assigned to any route longer than 400 km. All other bus types are banned from these routes.
  + **Minimum Occupancy Floor:** Any route with an occupancy rate below 30% (< 0.3) is strictly forbidden. The model is forced to assign 0 trips to these routes.
  + **Bus Trip Limit:** Each individual bus is limited to running a maximum of 10 total trips, regardless of which routes it takes. This prevents over-utilization of a single vehicle.

**Solving and Analysis**

* **Solve:** The model is instructed to find the best possible solution within a timeLimit=300 seconds (5 minutes). This prevents the script from running indefinitely on a very complex problem.
* **Results:** The script gathers all the bus-routes that were assigned one or more trips into a new DataFrame, optimized\_routes2.
* **Insights:** It then prints a summary of the solution, including:
  + Total Net Profit achieved.
  + Total Fuel Used by the optimized schedule.
  + The total number of buses used compared to the total available.
  + The Top 3 most profitable routes.
  + The single **most fuel-efficient route** (based on profit-per-liter).

## Code

# Model Setup  
  
model = LpProblem("Bus\_Route\_Optimization", LpMaximize)  
x = {r: LpVariable(r, lowBound=0, upBound=3, cat='Integer') for r in df['var\_name']}  
  
  
  
# Net profit  
df['net\_profit'] = df['profit'] - df['total\_fuel\_cost']  
  
# Adjust net profit for low occupancy  
df['adjusted\_net\_profit'] = df.apply(  
 lambda r: r['net\_profit'] \* 0.8 if r['occupancy\_rate'] < 0.5 else r['net\_profit'], axis=1  
)  
  
# Seat efficiency bonus  
df['seat\_efficiency'] = df['occupancy\_rate'] \* df['capacity']  
  
# Weighted objective: profit, fuel minimization, seat efficiency  
alpha = 1.0 # net profit weight  
beta = 0.5 # fuel minimization weight  
gamma = 0.2 # seat efficiency weight  
  
df['weighted\_obj'] = (alpha\*df['adjusted\_net\_profit']  
 - beta\*df['fuel\_consumed\_liters']  
 + gamma\*df['seat\_efficiency'])  
  
  
objective\_map = df.set\_index('var\_name')['weighted\_obj'].to\_dict()  
model += lpSum(x[r] \* objective\_map[r] for r in x), "Weighted\_MultiObjective"  
  
  
# Constraints  
  
# Depot limit:  
for depot in df['depot'].unique():  
 depot\_routes = df[df['depot']==depot]['var\_name']  
 model += lpSum(x[r] for r in depot\_routes) <= 180, f"DepotLimit\_{depot}"  
  
# Long routes (>400 km) only for Volvo buses  
long\_routes\_mask = (df['distance\_km'] > 400) & (df['bus\_type'].str.lower() != 'volvo')  
for r in df.loc[long\_routes\_mask, 'var\_name']:  
 model += x[r] == 0, f"LongRoute\_{r}"  
  
# Minimum occupancy (40%)  
min\_occupancy = 0.4  
low\_occ\_mask = df['occupancy\_rate'] < min\_occupancy  
for r in df.loc[low\_occ\_mask, 'var\_name']:  
 model += x[r] == 0, f"MinOccupancy\_{r}"  
  
# Max distance per bus per day  
max\_distance\_per\_bus = 3000  
for \_, row in df.iterrows():  
 model += x[row['var\_name']] \* row['distance\_km'] <= max\_distance\_per\_bus, f"MaxDistance\_{row['var\_name']}"  
  
  
  
# Solve  
  
print("\n Solving optimization model...")  
model.solve(PULP\_CBC\_CMD(msg=True))  
  
  
# Extract Results  
  
results = []  
for \_, row in df.iterrows():  
 trips = x[row['var\_name']].value()  
 if trips and trips > 0:  
 results.append({  
 'bus\_id': row['bus\_id'],  
 'route': row['route'],  
 'depot': row['depot'],  
 'bus\_type': row['bus\_type'],  
 'occupancy\_rate': row['occupancy\_rate'],  
 'distance\_km': row['distance\_km'],  
 'adjusted\_net\_profit': row['adjusted\_net\_profit'],  
 'trips\_selected': trips,  
 'total\_net\_profit': row['adjusted\_net\_profit'] \* trips,  
 'fuel\_consumed\_liters': row['fuel\_consumed\_liters'],  
 'total\_fuel\_used': row['fuel\_consumed\_liters'] \* trips,  
 'seat\_efficiency': row['seat\_efficiency']  
 })  
  
optimized\_routes3 = pd.DataFrame(results)  
  
  
# Display Results (Profit before Fuel)  
  
display\_columns = [  
 'bus\_id', 'route', 'depot', 'bus\_type', 'occupancy\_rate', 'distance\_km',  
 'adjusted\_net\_profit', 'trips\_selected', 'total\_net\_profit',  
 'fuel\_consumed\_liters', 'total\_fuel\_used', 'seat\_efficiency'  
]  
optimized\_routes\_display = optimized\_routes3[display\_columns]  
  
print("\nOptimized Routes (Net Profit First):")  
print(optimized\_routes\_display)  
  
print("\n Total Optimized Net Profit:", round(optimized\_routes\_display['total\_net\_profit'].sum(), 2))  
print(" Total Fuel Consumed:", round(optimized\_routes\_display['total\_fuel\_used'].sum(), 2))  
  
  
# Insights  
  
  
if not optimized\_routes3.empty:  
 print("\n Insights Summary:")  
 print("=" \* 50)  
  
 # Key Metrics  
 total\_profit = optimized\_routes3['total\_net\_profit'].sum()  
 total\_fuel = optimized\_routes3['total\_fuel\_used'].sum()  
 avg\_occ = optimized\_routes3['occupancy\_rate'].mean()  
 avg\_distance = optimized\_routes3['distance\_km'].mean()  
 avg\_seat\_eff = optimized\_routes3['seat\_efficiency'].mean()  
  
 print(f" Total Net Profit: ₹{round(total\_profit, 2)}")  
 print(f" Total Fuel Used: {round(total\_fuel, 1)} liters")  
 print(f" Average Seat Efficiency: {round(avg\_seat\_eff, 2)}")  
 print(f" Average Occupancy Rate: {round(avg\_occ \* 100, 1)}%")  
 print(f" Average Route Distance: {round(avg\_distance, 1)} km")  
  
  
 # Top 3 Profitable Routes  
 top3 = optimized\_routes3.nlargest(3, 'total\_net\_profit')  
 print("\n Top 3 Most Profitable Routes:")  
 for \_, r in top3.iterrows():  
 print(f" • {r['route']} ({r['bus\_type']}) → ₹{round(r['total\_net\_profit'], 2)}")  
  
 # Depot Utilization  
 depot\_counts = optimized\_routes3['depot'].value\_counts()  
 print("\n Depot Utilization Summary:")  
 for depot, count in depot\_counts.items():  
 print(f" • {depot}: {count} optimized routes")  
  
  
 # Most Fuel-Efficient Route  
 fuel\_eff = optimized\_routes3.copy()  
 fuel\_eff['profit\_per\_litre'] = fuel\_eff['total\_net\_profit'] / fuel\_eff['total\_fuel\_used']  
 efficient = fuel\_eff.nlargest(1, 'profit\_per\_litre').iloc[0]  
  
 print(f"\n Most Fuel-Efficient Route:")  
 print(f" • {efficient['route']} ({efficient['bus\_type']}) ")  
  
  
 # Highest Seat Efficiency  
 top\_seat = optimized\_routes3.nlargest(1, 'seat\_efficiency').iloc[0]  
 print(f"\n Highest Seat Efficiency Route:")  
 print(f" • {top\_seat['route']} ({top\_seat['bus\_type']})")  
  
  
else:  
 print("\n No optimization results found. Please check model constraints or weights.")

Output:

Total Optimized Net Profit: 7691863.18  
 Total Fuel Consumed: 53536.71  
  
 Insights Summary:  
==================================================  
 Total Net Profit: ₹7691863.18  
 Total Fuel Used: 53536.7 liters  
 Average Seat Efficiency: 39.46  
 Average Occupancy Rate: 77.0%  
 Average Route Distance: 281.1 km  
  
 Top 3 Most Profitable Routes:  
 • Guntur-Hyderabad (Volvo AC) → ₹102893.93  
 • Kadapa-Hyderabad (Volvo AC) → ₹102283.48  
 • Ongole-Hyderabad (Super Luxury) → ₹102131.49  
  
 Depot Utilization Summary:  
 • Guntur: 50 optimized routes  
 • Kurnool: 47 optimized routes  
 • Visakhapatnam: 44 optimized routes  
 • Vijayawada: 43 optimized routes  
 • Nellore: 40 optimized routes  
 • Tirupati: 37 optimized routes  
 • Hyderabad: 31 optimized routes  
  
 Most Fuel-Efficient Route:  
 • Vijayawada-Tirupati (Volvo AC)   
  
 Highest Seat Efficiency Route:  
 • Hyderabad-Vijayawada (Ordinary)

**Explanation**:

This script defines the most complex optimization model of the three, moving to a **weighted multi-objective** goal. It aims to find the best schedule by simultaneously:

1. Maximizing net profit.
2. Minimizing fuel consumption (with a heavy penalty).
3. Rewarding "seat efficiency" (filling up larger buses).

It also introduces new, more granular constraints at the individual bus-route level.

**1. Data Preparation and Feature Engineering**

This section cleans the data and creates the new metrics needed for the objective function.

* **Standard Cleaning:** As before, all column names are standardized (lowercase, stripped, underscored), and key columns like profit, fuel\_consumed\_liters, capacity, and passengers are converted to numeric types. occupancy\_rate is normalized to a 0-1 proportion.
* **Net Profit:** Calculates net\_profit (profit - total\_fuel\_cost) as the baseline profitability metric.
* **Adjusted Net Profit:** Creates adjusted\_net\_profit. A 20% profit penalty (\* 0.8) is applied to any route with an occupancy rate below 50%.
* **Seat Efficiency (New Metric):** A new metric, seat\_efficiency, is created by calculating occupancy\_rate \* capacity. This is a crucial new feature. It measures the *actual number of seats filled* rather than just the percentage.

**2. The Multi-Objective Function**

This is the core of the model and defines its complex trade-offs. The goal is to maximize a weighted\_obj score, which is a blend of three different business goals.

The score for each route is calculated as:

[ (1.0 \* Adjusted\_Net\_Profit) ] - [ (0.5 \* Fuel\_Consumed\_Liters) ] + [ (0.2 \* Seat\_Efficiency) ]

Let's break down the components:

* **alpha = 1.0 (Profit):** The model gets 100% of the value from the adjusted\_net\_profit. This remains the primary driver.
* **beta = 0.5 (Fuel Penalty):** For every liter of fuel a route consumes, **0.5 points are subtracted** from its score. This is a significant penalty (twice as high as in the previous model), signaling a much stronger business desire to minimize fuel consumption.
* **gamma = 0.2 (Seat Bonus):** For every "efficient seat" (as calculated above), **0.2 points are added** to the score. This creates a new incentive for the model to not just pick profitable routes, but to pick routes where it can fill many seats.

**3. Decision Variables & Constraints**

The model must maximize the total weighted\_obj score while adhering to a strict set of rules.

* **Decision Variable (x):** The model decides how many trips to assign to each bus-route. This is an integer between **0 and 3** (upBound=3). Note that this is more restrictive than the 0-5 range in the second model.
* **Constraints (The Rules):**
  + **Fixed Depot Limit:** Each depot is given a **fixed cap of 180 total trips**. This is a simpler, non-dynamic constraint, unlike the one in the previous model.
  + **Long Route Restriction:** Only 'Volvo' buses are allowed to run routes longer than 400 km.
  + **Minimum Occupancy Floor:** A **hard floor of 40% occupancy** (min\_occupancy = 0.4) is set. Any route that averages less than 40% occupancy is banned (x[r] == 0), regardless of its profitability.
  + **Max Distance per Bus-Route (New Constraint):** This is a new, highly specific operational constraint. The *total distance* for any *single bus-route assignment* cannot exceed 3000 km.
    - **Formula:** (Trips Assigned) \* (Route Distance) <= 3000
    - **Example:** If a route is 1,200 km long, the model can only assign it a maximum of 2 trips (2 \* 1200 = 2400), because 3 trips would violate the constraint (3 \* 1200 = 3600). This simulates a realistic limit on maintenance or driver fatigue for a single route.

**4. Solving and Analysis**

* **Solve:** The model is solved to find the combination of trips that gives the highest possible total weighted\_obj score.
* **Results:** The final selected routes (where trips > 0) are extracted into the optimized\_routes3 DataFrame.
* **Insights:** The analysis is upgraded to reflect the new, complex objective. It reports on:
  + Total Net Profit and Total Fuel Used.
  + **Average Seat Efficiency:** A new key metric to see if the model successfully prioritized filling larger buses.
  + Top 3 Routes by Profit.
  + Most Fuel-Efficient Route (profit-per-liter).
  + **Highest Seat Efficiency Route:** A new insight that directly identifies the single best route according to the new seat\_efficiency metric.

## Code

!pip install xlsxwriter

**Explanation**  
This command installs the **xlsxwriter** library, which is a Python module used for **creating and writing to new Excel files** in the .xlsx format.

It's commonly used when you need to:

* Export a pandas DataFrame to an Excel file (df.to\_excel(..., engine='xlsxwriter')).
* Create custom Excel reports with charts, formatting (like colors and bold text), and multiple sheets.

## Code

# Saving All Optimization In Single Excel File  
  
# File name  
output\_file = "all\_optimized\_routes.xlsx"  
  
# Save multiple DataFrames to different sheets  
with pd.ExcelWriter(output\_file, engine='xlsxwriter') as writer:  
 optimized\_routes1.to\_excel(writer, sheet\_name='Optimized\_Routes1', index=False)  
 optimized\_routes2.to\_excel(writer, sheet\_name='Optimized\_Routes2', index=False)  
 optimized\_routes3.to\_excel(writer, sheet\_name='Optimized\_Routes3', index=False)  
  
print(f"\n All optimized results saved to '{output\_file}' with multiple sheets.")

Output:

All optimized results saved to 'all\_optimized\_routes.xlsx' with multiple sheets.

**Explanation:**

This code block is used to **bundle all three of your optimization result DataFrames** into a single, organized Excel file.

Instead of saving three separate files ,it uses a more powerful feature from pandas called ExcelWriter.

Here’s the breakdown:

* **pd.ExcelWriter(output\_file, ...):** Think of this as opening a blank Excel workbook (named "all\_optimized\_routes.xlsx") in the background and getting it ready to receive data. We assign this open workbook to the variable writer.
* **engine='xlsxwriter':** This tells pandas to use the specific xlsxwriter library as the "engine" to build the file. This engine is great at creating new, complex Excel files.
* **with ... as writer::** This is a "context manager." It's a clean and safe way to work with files. It automatically **saves and closes** the Excel file as soon as the indented code block is finished, even if an error happens.
* **.to\_excel(writer, ...):** Inside the block, you call .to\_excel() on each of your DataFrames (optimized\_routes1, optimized\_routes2, optimized\_routes3).
* **sheet\_name=...:** This is the key. By passing the writer object (instead of a filename) and specifying a sheet\_name, you're telling pandas: "Don't create a new file; just add a new tab *inside* the file this writer is managing."
* **index=False:** This is a formatting choice. It prevents pandas from writing the DataFrame's row index (0, 1, 2, etc.) as the first column in the Excel sheet, which makes the final report look cleaner.

The final result is one .xlsx file with three distinct sheets ("Optimized\_Routes1", "Optimized\_Routes2", "Optimized\_Routes3"), making it much easier to share and compare the results of your different models.

**PowerBI DashBoard**

