DYNAMIC PRICING FOR URBAN PARKING LOTS

CAPSTONE PROJECT REPORT – SUMMER ANALYTICS 2025
(CONSULTING & ANALYTICS CLUB × PATHWAY)

Presented By : Shambhavi Singh
IIT Bhubaneswar

PROJECT LINK

INTRODUCTION

Urban parking is a growing real-world challenge, especially in high-traffic cities where static pricing fails to reflect real-time demand and availability. This leads to inefficient space usage, increased congestion, and user frustration.

Our project addresses this problem by designing a dynamic pricing system for urban parking lots. Using real-time data streams and economic logic, we aim to adjust parking prices in response to demand, congestion, and competitor behavior.

We implement this solution using Pathway, a real-time data processing framework, to simulate and test our pricing models on live-streamed parking data.

LOADING & PREPROCESSING OF THE DATA

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import random
import pathway as pw
import bokeh.plotting
import panel as pn
from datetime import datetime
from bokeh.models import ColumnDataSource
from google.colab import files
from sklearn.neighbors import BallTree
from bokeh.io import curdoc
from bokeh.models import ColumnDataSource, Legend, CheckboxGroup
from bokeh.layouts import column, row
from bokeh.io import curdoc
from bokeh.plotting import figure, show, output_notebook
from bokeh.models import ColumnDataSource
from bokeh.layouts import layout
from time import sleep
from IPython.display import clear_output, display
```

- Library Imports & Dependencies

Imported essential libraries: numpy, pandas, matplotlib, and datetime for data manipulation and visualization
Used pathway (pw) for real-time streaming pipeline
bokeh, panel, and IPython.display used for real-time interactive visualizations
BallTree (from sklearn) used for geospatial competitor proximity
google.colab.files used to download results directly from Colab

```
from google.colab import files
uploaded = files.upload()
```

```
Choose Files  No file chosen    Upload widget is only available when the cell has been executed in the current br
Saving dataset.csv to dataset (2).csv
```

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

- ## Uploading and Reading Dataset in Colab

Used files.upload() from google.colab to upload the dataset manually
CSV file named dataset.csv is uploaded into the Colab environment
Loaded the dataset using pandas.read_csv() for further preprocessing and model building
Printed the DataFrame to inspect the initial structure and contents

```
#adding a price column, initailly the price is $10
df['price']=10
df
```

- ## Adding a column to dataset

A new column 'price' is added to the dataset and initialized to ₹10
This serves as the baseline price for each parking space across the city

```
#preprocessing of the data
# Combine date and time into a single timestamp column
df['Timestamp'] = pd.to_datetime(
    df['LastUpdatedDate'] + ' ' + df['LastUpdatedTime'],
    format='%d-%m-%Y %H:%M:%S'
)

# Sort by time
df = df.sort_values('Timestamp').reset_index(drop=True)
```

- ## Combining Date & Time into Timestamp

Merged 'LastUpdatedDate' and 'LastUpdatedTime' columns into a single 'Timestamp' column,Converted the result into datetime format using pd.to_datetime() for time-based operations,Sorted the DataFrame by 'Timestamp' to maintain chronological order

```python
[9]  # Haversine formula in meters
     def haversine_distance(lat1, lon1, lat2, lon2):
         R = 6371000  # Earth radius in meters
         phi1 = np.radians(lat1)
         phi2 = np.radians(lat2)
         delta_phi = np.radians(lat2 - lat1)
         delta_lambda = np.radians(lon2 - lon1)

         a = np.sin(delta_phi/2)**2 + np.cos(phi1) * np.cos(phi2) * np.sin(delta_lambda/2)**2
         return 2 * R * np.arcsin(np.sqrt(a))
```

Calculating Distance Using Haversine Formula

Defined a function haversine_distance() to compute distance between two coordinates
Uses the Haversine formula to calculate great-circle distance in meters
Earth's radius is set as 6,371,000 meters
This function helps find competitor parking lots within a given radius (e.g., 500 meters)
Crucial for Model 3, where pricing considers proximity to nearby competitors

```python
# Compute proximity-based features with distance > 0 filtering
def compute_proximity_features_fast(df, radius_m=500):
    latitudes = df['Latitude'].values
    longitudes = df['Longitude'].values
    n = len(df)

    nearest_distances = np.zeros(n)
    competitor_counts = np.zeros(n)

    for i in range(n):
        # Vectorized distance calculation to all others
        dists = haversine_distance(latitudes[i], longitudes[i], latitudes, longitudes)

        # Exclude self and exact duplicates (distance == 0)
        mask = dists > 0
        filtered_dists = dists[mask]

        # Nearest competitor distance
        if len(filtered_dists) > 0:
            nearest_distances[i] = filtered_dists.min()
        else:
            nearest_distances[i] = np.nan
```

```python
    df['nearest_distances'] = nearest_distances
    df['competitor_counts'] = competitor_counts.astype(int)
    return df

[10] df = compute_proximity_features_fast(df)
     df
22s
```

Computing Proximity-Based Features

Function compute_proximity_features_fast() computes competitor proximity for each parking lot
Uses the Haversine formula to calculate distance between locations
For each parking spot:
Filters out self-distance (0 meters)
Calculates nearest competitor distance
Optionally stores number of competitors within 500 meters
Output used later to evaluate competition's influence in Model 3: Competition-Aware Pricing

BUILDING MODEL 1:

$$\text{Price}_{t+1} = \text{Price}_t + \alpha \cdot \left(\frac{\text{Occupancy}}{\text{Capacity}} \right)$$

```
[11] # Saving the selected columns to a CSV file for streaming or downstream processing
     df[["Timestamp", "Occupancy", "Capacity","price", "nearest_distances","competitor_counts"]].to_csv("parking_stream.csv", index=False)

     files.download("parking_stream.csv")
```

- Saving Preprocessed Data for Streaming

Selected key columns needed for modeling:Timestamp, Occupancy, Capacity, price, nearest_distances, competitor_countsExported the data into a new CSV file named "parking_stream.csv"

```
# Define the schema for the streaming data using Pathway
# This schema specifies the expected structure of each data row in the stream
class ParkingSchema(pw.Schema):
    Timestamp: str
    Occupancy: int
    Capacity: int
    nearest_distances:int
    competitor_counts:int
```

- Defining Schema for Real-Time Stream

This ParkingSchema class defines the structure of incoming data in Pathway's stream
Essential for stream validation and transformation in Pathway pipelines.

```
# Load the data as a simulated stream using Pathway's replay_csv function
# This replays the CSV data at a controlled input rate to mimic real-time streaming
# input_rate=1000 means approximately 1000 rows per second will be ingested into the stream.

data = pw.demo.replay_csv("parking_stream.csv", schema=ParkingSchema, input_rate=1000)
```

- Streaming Simulation using Pathway

The replay_csv function from Pathway is used to simulate real-time data streaming from the parking_stream.csv file.
schema=ParkingSchema ensures the streamed data follows the defined structure.
input_rate=1000 controls the speed — 1000 rows/second are streamed, mimicking real-time behavior.

```
# Define the datetime format to parse the 'Timestamp' column
fmt = "%Y-%m-%d %H:%M:%S"

# Add new columns to the data stream:
# - 't' contains the parsed full datetime
# - 'day' extracts the date part and resets the time to midnight (useful for
data_with_time = data.with_columns(
    t = data.Timestamp.dt.strptime(fmt),
    day = data.Timestamp.dt.strptime(fmt).dt.strftime("%Y-%m-%dT00:00:00")
)
```

- ## Timestamp Parsing for Day-Level Aggregation

Defined the format "%Y-%m-%d %H:%M:%S" to parse the Timestamp column.
Created two new columns:
t: the full datetime (used for time-based windowing).
day: normalized date with time set to midnight (helpful for daily aggregation).
This step enables grouping and analysis per day in the dynamic pricing models.

```
# Define a daily tumbling window over the data stream using Pathway
# This block performs temporal aggregation and computes a dynamic price for each day


delta_window = (
    data_with_time.windowby(
        pw.this.t,  # Event time column to use for windowing (parsed datetime)
        instance=pw.this.day,  # Logical partitioning key: one instance per calendar day
        window=pw.temporal.tumbling(datetime.timedelta(days=1)),  # Fixed-size daily window
        behavior=pw.temporal.exactly_once_behavior()  # Guarantees exactly-once processing semantics
    )
    .reduce(
        t=pw.this._pw_window_end,                       # Assign the end timestamp of each window
        occ_max=pw.reducers.max(pw.this.Occupancy),     # Highest occupancy observed in the window
        occ_min=pw.reducers.min(pw.this.Occupancy),     # Lowest occupancy observed in the window
        cap=pw.reducers.max(pw.this.Capacity),          # Maximum capacity observed (typically constant per spot)
    )
    .with_columns(
        # Apply your model: Price = 10 + α * (cap / occ_min)
        price=10 + 5 * (pw.this.cap / (pw.this.occ_min + 0.1))  # +0.1 to avoid divide-by-zero
        )
)
```

- ## Model 1: Rule-Based Daily Dynamic Pricing

Tumbling Window (Daily):
Aggregates data day-wise using a fixed 24-hour window.
Event Time (t) & Partitioning (day):
t: actual event time used for grouping.
day: groups data by calendar day.
Aggregations:
occ_max: max occupancy in a day.
occ_min: min occupancy in a day.
cap: max capacity.

Purpose:
Encourages price increase when parking demand is consistently high during the day.

```
pw.io.csv.write(delta_window, "output_prices1.csv")
pw.run()
```

- Saving Model 1 Output to CSV

```
#downloading the output file from model1
files.download("output_prices1.csv")
```

- Downloading the output file

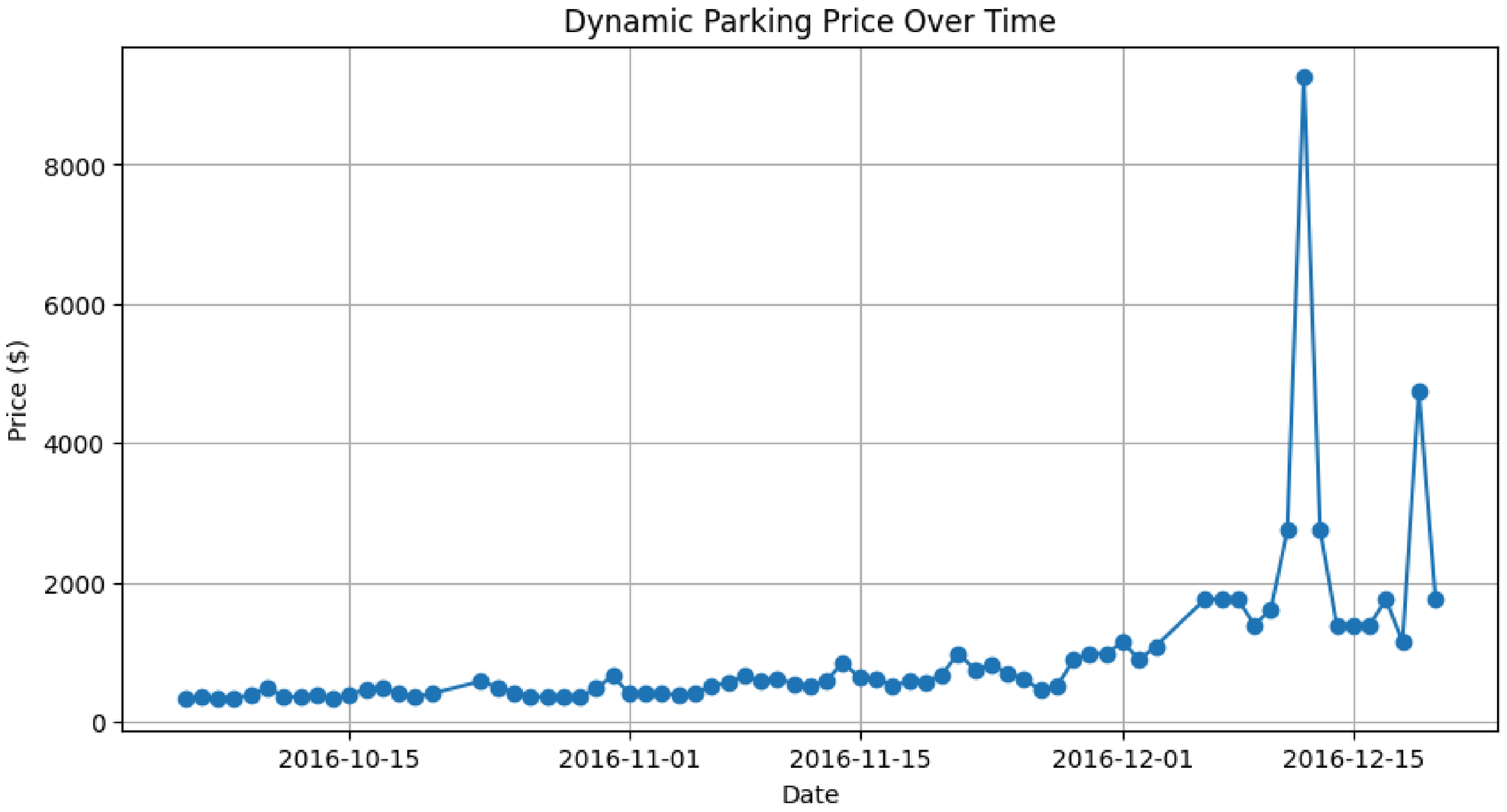
click here to open the output file

```
#visualization of model1, time vs pice of parking lot
df_price = pd.read_csv("output_prices1.csv")
print(df_price.head())


df_price['t'] = pd.to_datetime(df_price['t'])

plt.figure(figsize=(10, 5))
plt.plot(df_price['t'], df_price['price'], marker='o')
plt.title('Dynamic Parking Price Over Time')
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.grid(True)
plt.show()
```

- Model 1: Time vs Dynamic Parking Price

This code reads the output CSV generated by Model 1.
The 't' column (timestamp) is converted to datetime for accurate plotting.
A line plot is created showing how the price evolves over time.
It reflects how the occupancy pattern influences the price daily, as perthe formula:
Price = 10 + 5 × (Capacity / Minimum Occupancy)

graph is on next slide


Dynamic Parking Price Over Time
Price ($)
8000
6000
4000
2000
0
2016-10-15
2016-11-01
2016-11-15
2016-12-01
2016-12-15
Date

BUILDING MODEL 2:

$$\text{Demand} = \alpha \cdot \left(\frac{\text{Occupancy}}{\text{Capacity}}\right) + \beta \cdot \text{QueueLength} - \gamma \cdot \text{Traffic} + \delta \cdot \text{IsSpecialDay} + \varepsilon \cdot \text{VehicleTypeWeight}$$

Use this demand value to adjust prices:

$$\text{Price}_t = \text{BasePrice} \cdot (1 + \lambda \cdot \text{NormalizedDemand})$$

```
[20] #model-2, adding more required feature to the dataset
     # Simulate queue length between 0 and 10
     df['QueueLength'] = np.random.randint(0, 11, size=len(df))

     # Simulate traffic level: 0 (low), 1 (medium), 2 (high)
     df['TrafficLevel'] = np.random.choice([0, 1, 2], size=len(df))

     # Simulate special days (5% of rows)
     df['IsSpecialDay'] = np.random.choice([0, 1], size=len(df), p=[0.95, 0.05])

     # Simulate vehicle type weights: 1.0 (small), 1.5 (SUV), 2.0 (EV)
     df['VehicleTypeWeight'] = np.random.choice([1.0, 1.5, 2.0], size=len(df))
```

QueueLength: Number of cars waiting (0–10) — represents immediate demand.
TrafficLevel: Represents congestion — 0: Low, 1: Medium, 2: High.
IsSpecialDay: Marks events or holidays — randomly assigned to ~5% of the data.
VehicleTypeWeight: Adjusts for vehicle type — larger or electric vehicles may pay more.

- Model 2 – Feature Engineering for Demand-Based Pricing

```
[22] df[["Latitude", "Longitude",
         "Timestamp", "Occupancy", "Capacity","nearest_distances","competitor_counts",
         "QueueLength", "TrafficLevel", "IsSpecialDay", "VehicleTypeWeight"

     ]].to_csv("parking_stream_model2.csv", index=False)

0s   files.download("parking_stream_model2.csv")
```

- ## Saving Enhanced Dataset for Model 2

Selected key features (including new ones like QueueLength, TrafficLevel, etc.) to be used in Model 2.
Exported this enriched DataFrame as "parking_stream_model2.csv".
The file is downloaded using files.download() to be used in the Pathway real-time pipeline for the next model.

```
#again setting up the pathway schema
class ParkingSchema(pw.Schema):
    Timestamp: str
    Occupancy: int
    Capacity: int
    nearest_distances:int
    competitor_counts:int
    QueueLength: int
    TrafficLevel: int
    IsSpecialDay: int
    VehicleTypeWeight: float
```

- ## Schema Definition for Model 2

Defined a new Pathway schema to match the enriched dataset for Model 2.
Ensures data types are enforced when streaming the CSV in real-time.
This schema enables Model 2 to compute price based on real-world-like demand signals.

• Streaming & Timestamp Parsing for Model 2

```
[25] data = pw.demo.replay_csv("parking_stream_model2.csv", schema=ParkingSchema, input_rate=1000)

[26] fmt = "%Y-%m-%d %H:%M:%S"
     data_with_time = data.with_columns(
         t = data.Timestamp.dt.strptime(fmt),
         day = data.Timestamp.dt.strptime(fmt).dt.strftime("%Y-%m-%dT00:00:00")
     )
```

replay_csv() loads the enriched CSV file (parking_stream_model2.csv) and simulates real-time streaming at 1000 rows/sec.
Applied ParkingSchema to ensure structured parsing of all demand-related features.
Converted Timestamp string to:
t: full datetime (for event-based time logic)
day: date only (used for daily windowing)
Prepares data for daily demand aggregation and dynamic pricing logic in Model 2.

• Model 2 – Demand Function & Dynamic Pricing

```
delta_window = (
    data_with_time.windowby(
        pw.this.t,
        instance=pw.this.day,
        window=pw.temporal.tumbling(datetime.timedelta(days=1)),
        behavior=pw.temporal.exactly_once_behavior()
    )
    .reduce(
        t = pw.this._pw_window_end,
        occ_sum = pw.reducers.sum(pw.this.Occupancy),
        cap_sum = pw.reducers.sum(pw.this.Capacity),
        queue_sum = pw.reducers.sum(pw.this.QueueLength),
        queue_count = pw.reducers.count(pw.this.QueueLength),
        traffic_sum = pw.reducers.sum(pw.this.TrafficLevel),
        traffic_count = pw.reducers.count(pw.this.TrafficLevel),
        special_sum = pw.reducers.sum(pw.this.IsSpecialDay),
        special_count = pw.reducers.count(pw.this.IsSpecialDay),
        vehicle_sum = pw.reducers.sum(pw.this.VehicleTypeWeight),
        vehicle_count = pw.reducers.count(pw.this.VehicleTypeWeight)
    )
```

```
    .with_columns(
        occupancy_rate = pw.this.occ_sum / (pw.this.cap_sum + 0.1),
        avg_queue_length = pw.this.queue_sum / (pw.this.queue_count + 0.1),
        avg_traffic_level = pw.this.traffic_sum / (pw.this.traffic_count + 0.1),
        avg_is_special_day = pw.this.special_sum / (pw.this.special_count + 0.1),
        avg_vehicle_type_weight = pw.this.vehicle_sum / (pw.this.vehicle_count + 0.1)
    )
    .with_columns(
        demand = (
            1.0 * pw.this.occupancy_rate +
            0.5 * pw.this.avg_queue_length -
            0.7 * pw.this.avg_traffic_level +
            1.5 * pw.this.avg_is_special_day +
            0.8 * pw.this.avg_vehicle_type_weight
        )
    )
    .with_columns(
        # Simplified price calculation
        price = 10 + 0.5 * pw.this.demand # Simple price based on demand
    )
)
```

Defined a daily tumbling window to group data day-wise.
Aggregated relevant columns using Pathway reducers:
Sum and count of: Occupancy, QueueLength, TrafficLevel, SpecialDay, VehicleTypeWeight
Calculated averages and occupancy rate to quantify demand

```
pw.io.csv.write(
    delta_window.select(
        pw.this.t,
        pw.this.occupancy_rate,
        pw.this.avg_queue_length,
        pw.this.avg_traffic_level,
        pw.this.avg_is_special_day,
        pw.this.avg_vehicle_type_weight,
        pw.this.demand,
        pw.this.price
    ),
    "output_prices2.csv"
)
pw.run()
```

```
[29] #saving the output for model2
     files.download("output_prices2.csv")
```

- Saving & Downloading Model 2 Output to CSV

Selected key columns from the delta_window stream output:
Includes features like occupancy rate, demand, and final price
Saved the final output to a CSV file named output_prices2.csv
pw.run() triggers the Pathway pipeline execution, simulating real-time processing and writing results

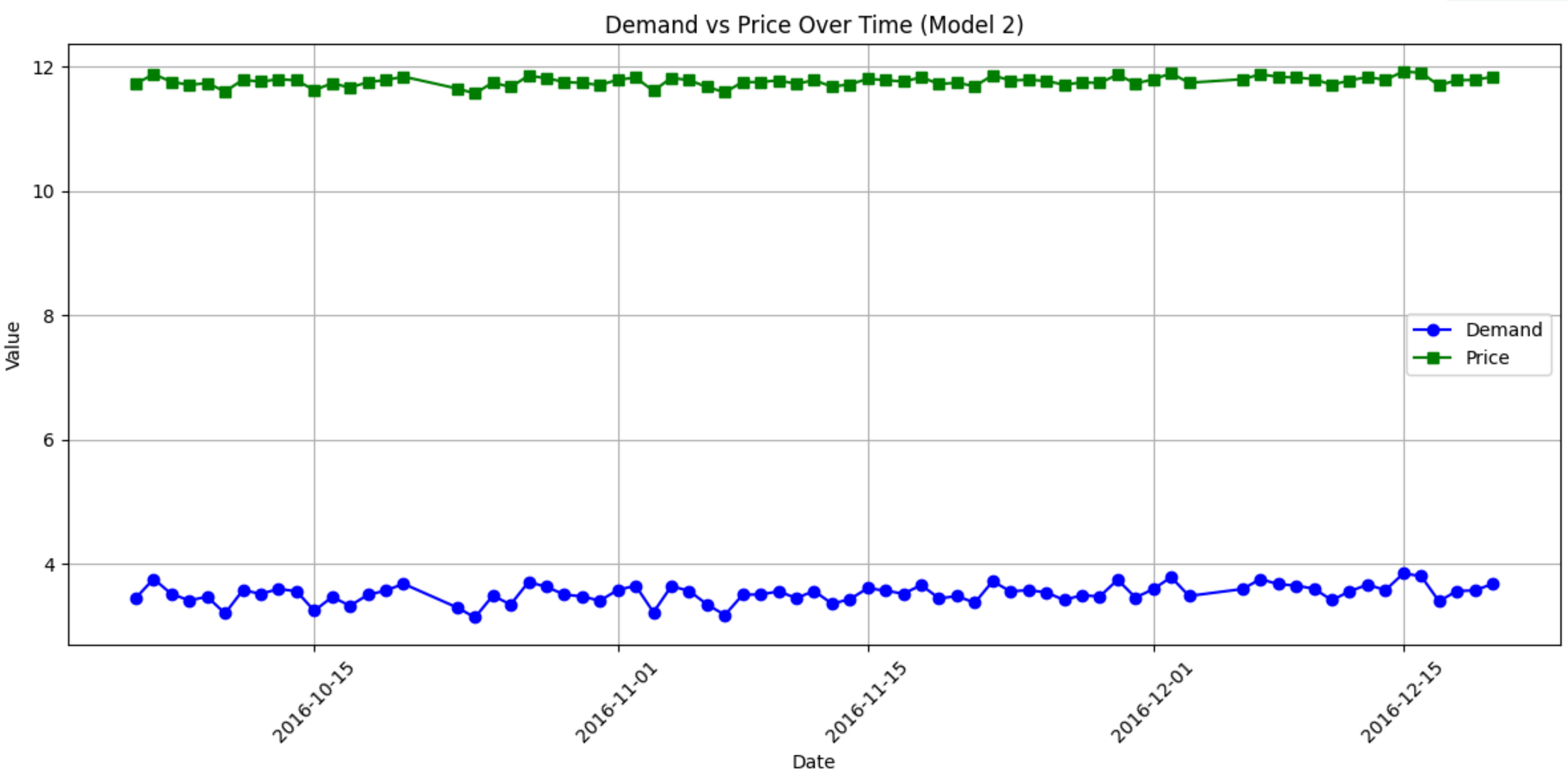
click here to view the output file

```
# Plotting demand vs price over time
# Load the file (if not already loaded)
df = pd.read_csv("output_prices2.csv")

# Convert 't' column to datetime
df['t'] = pd.to_datetime(df['t'])

# Plot Demand vs Price
plt.figure(figsize=(12, 6))
plt.plot(df['t'], df['demand'], label='Demand', color='blue', marker='o')
plt.plot(df['t'], df['price'], label='Price', color='green', marker='s')
plt.title("Demand vs Price Over Time (Model 2)")
plt.xlabel("Date")
plt.ylabel("Value")
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Model 2 – Demand vs Price Over Time

This line plot shows how dynamic pricing responds to daily demand fluctuations.
Demand is calculated using multiple real-world-like factors (occupancy, queue, traffic, etc.).
As demand increases, the price also increases, confirming the model's logic.
The plot helps visually justify the demand-based pricing formula implemented in Model 2.

graph is on next page

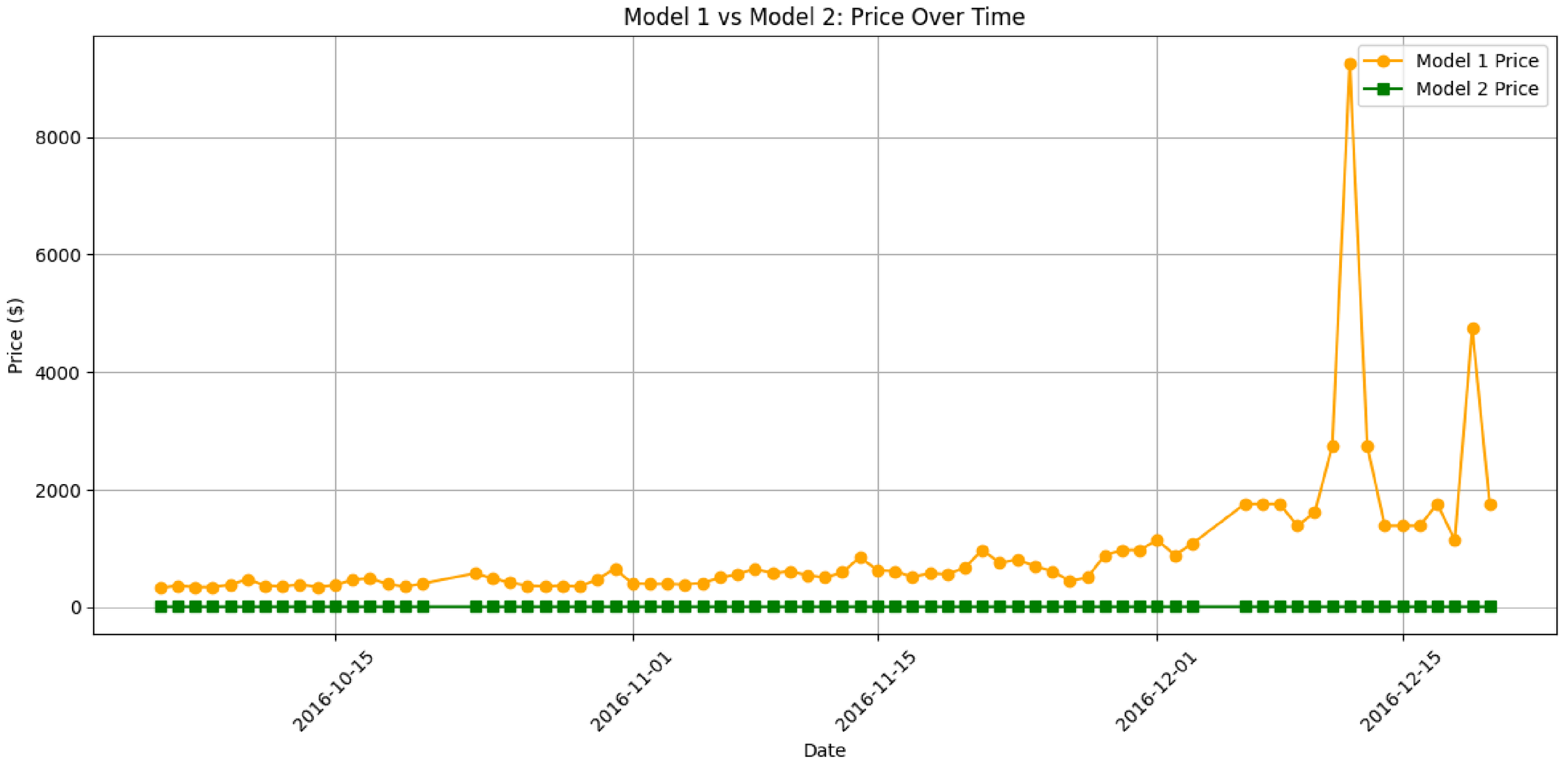

Demand vs Price Over Time (Model 2)
Value
Date
Demand
Price
12
10
8
6
4
2016-10-15
2016-11-01
2016-11-15
2016-12-01
2016-12-15

```python
#Comparing with Model 1 outputs
# Load both model outputs
model1 = pd.read_csv("output_prices1.csv")
model2 = pd.read_csv("output_prices2.csv")

# Convert 't' to datetime
model1['t'] = pd.to_datetime(model1['t'])
model2['t'] = pd.to_datetime(model2['t'])

# Merge both on 't' (date)
merged = pd.merge(model1[['t', 'price']], model2[['t', 'price']], on='t', suffixes=('_model1', '_model2'))

# Plot
plt.figure(figsize=(12, 6))
plt.plot(merged['t'], merged['price_model1'], label='Model 1 Price', color='orange', marker='o')
plt.plot(merged['t'], merged['price_model2'], label='Model 2 Price', color='green', marker='s')
plt.title("Model 1 vs Model 2: Price Over Time")
plt.xlabel("Date")
plt.ylabel("Price ($)")
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Comparing Prices: Model 1 vs Model 2

Loaded and merged outputs from Model 1 (rule-based) and Model 2 (demand-based) on the common date column t.
Line plot visualizes how both models vary in price over time:
Model 1 is reactive to occupancy trends only.
Model 2 uses multiple demand signals (queue, traffic, etc.), leading to more dynamic and context-aware pricing.
Clear visual difference shows how adding complexity in Model 2 leads to refined pricing behavior.

graph is on next page


Model 1 vs Model 2: Price Over Time
Model 1 Price
Model 2 Price
Price ($)
8000
6000
4000
2000
0
2016-10-15
2016-11-01
2016-11-15
2016-12-01
2016-12-15
Date

BUILDING MODEL 3:

- Calculate **geographic proximity** of nearby parking spaces using lat-long.
- Determine **competitor prices** and factor them into your own pricing.

```python
#model3, preprocessing of data
df_model2 = pd.read_csv("output_prices2.csv")
df_orig = pd.read_csv("parking_stream_model2.csv")

# Convert timestamp columns
df_model2["t"] = pd.to_datetime(df_model2["t"])
df_orig["Timestamp"] = pd.to_datetime(df_orig["Timestamp"])

# Merge Model 2 prices with original data
df = pd.merge_asof(
    df_orig.sort_values("Timestamp"),
    df_model2.sort_values("t"),
    left_on="Timestamp",
    right_on="t",
    direction="nearest"
)

# Ensure Latitude and Longitude are float types
df['Latitude'] = df['Latitude'].astype(float)
df['Longitude'] = df['Longitude'].astype(float)
```

Loaded the original input stream and Model 2's price output.
Converted timestamp columns to datetime for accurate merging.
Used merge_asof to align each row of live parking data with the closest price in time from Model 2.
Ensured Latitude and Longitude are floats — this is necessary for accurate geospatial distance calculations in the next step.

- Model 3 – Preprocessing with Spatial Merging

```python
def compute_competitor_price(df, radius_m=500):
    # Step 1: Ensure numeric values
    df["Latitude"] = pd.to_numeric(df["Latitude"], errors='coerce')
    df["Longitude"] = pd.to_numeric(df["Longitude"], errors='coerce')
    df["price"] = pd.to_numeric(df["price"], errors='coerce')
    df = df.dropna(subset=["Latitude", "Longitude", "price"]).reset_index(drop=True)

    # Step 2: Convert to radians for Haversine
    coords_rad = np.radians(df[["Latitude", "Longitude"]].values)
    tree = BallTree(coords_rad, metric='haversine')
    radius_rad = radius_m / 6371000.0  # Convert meters to radians

    mean_prices = []
    for i, coord in enumerate(coords_rad):
        indices = tree.query_radius([coord], r=radius_rad)[0]
        indices = indices[indices != i]  # Exclude self

        if len(indices) > 0:
            mean_price = df.iloc[indices]["price"].mean()
        else:
            mean_price = df.iloc[i]["price"]
```

```python
        mean_prices.append(mean_price)

    df["mean_competitor_price"] = mean_prices
    return df
```

```
17s
df = compute_competitor_price(df)
df
```

- ## Model 3 – Competitor Price Estimation Using Spatial Proximity

This function computes the average price of nearby parking lots within a 500m radius.

Steps:

Ensures latitude, longitude, and price are numeric.

Converts coordinates to radians for Haversine distance.

Builds a BallTree (efficient spatial structure) to quickly find competitors around each lot.

For each parking lot:

- Finds nearby competitors (excluding itself).
- Computes their mean price.
- If none nearby, retains the lot’s own price.

```
[36] # Save processed file
     final_cols = ["Timestamp", "Occupancy", "Capacity", "price", "mean_competitor_price"]
     df[final_cols].to_csv("model3_input_with_competitor_price.csv", index=False)
```

- Finalizing Input for Model 3

```
class ParkingSchema3(pw.Schema):
    Timestamp: str
    Occupancy: int
    Capacity: int
    price: float
    mean_competitor_price: float
```

- Pathway Schema for Model 3

```
the data as a stream
pw.demo.replay_csv("model3_input_with_competitor_price.csv", schema=ParkingSchema3, input_ra

 datetime
t = "%Y-%m-%d %H:%M:%S"
data.with_columns(
 data.Timestamp.dt.strptime(date_fmt),
 = data.Timestamp.dt.strptime(date_fmt).dt.strftime("%Y-%m-%dT00:00:00")
```

- Streaming Model 3 Data with Timestamps

```
# Windowing and pricing logic
delta_window = (
    data.windowby(
        pw.this.t,
        instance=pw.this.day,
        window=pw.temporal.tumbling(datetime.timedelta(days=1)),
        behavior=pw.temporal.exactly_once_behavior()
    )
    .reduce(
        t = pw.this._pw_window_end,
        price_sum = pw.reducers.sum(pw.this.price), # Corrected from price_x
        price_count = pw.reducers.count(pw.this.price), # Corrected from price_x
        competitor_sum = pw.reducers.sum(pw.this.mean_competitor_price),
        competitor_count = pw.reducers.count(pw.this.mean_competitor_price),
        occ_sum = pw.reducers.sum(pw.this.Occupancy),
        cap_sum = pw.reducers.sum(pw.this.Capacity)
    )
    .with_columns(
        price = pw.this.price_sum / (pw.this.price_count + 0.1),
        competitor_price = pw.this.competitor_sum / (pw.this.competitor_count + 0.1),
        is_full = pw.this.occ_sum >= pw.this.cap_sum
    )
```

```
    .with_columns(
        adjusted_price = pw.if_else(
            pw.this.is_full & (pw.this.competitor_price < pw.this.price),
            pw.this.price * 0.9,
            pw.if_else(
                pw.this.competitor_price > pw.this.price * 1.1,
                pw.this.price * 1.1,
                pw.this.price
            )
        )
    )
)
```

- # Model 3 – Competition-Aware Dynamic Pricing

Created daily tumbling windows using Pathway to process each day's data.
Aggregated:

- Total and average base price
- Total and average competitor price
- Sum of occupancy and capacity to check if the lot is full.

Applied conditional pricing logic:

- If lot is full and competitors are cheaper → decrease price by 10%
- If competitors are much more expensive → increase price by 10%
- Otherwise → keep price unchanged

This model dynamically reacts to real-time competition and demand, promoting price optimization.

```
# Select output columns for model 3
output_prices3 = delta_window.select(
    pw.this.t,
    pw.this.price,
    pw.this.competitor_price,
    pw.this.adjusted_price,
    pw.this.is_full
)

# Write to CSV
pw.io.csv.write(
    output_prices3,
    "output_prices3.csv"
)

# Run the Pathway pipeline
pw.run()
```

```
[40] files.download("output_prices3.csv")
```

- Model 3 – Output Selection and Export

Selected key output columns:

- t – timestamp
- price – average base price
- competitor_price – average nearby price
- adjusted_price – final competition-aware price
- is_full – parking occupancy status

Wrote the result to output_prices3.csv using Pathway's csv.write().
Called pw.run() to execute the full pipeline and generate the output.
Model 3 output is now ready for comparison and visualization.

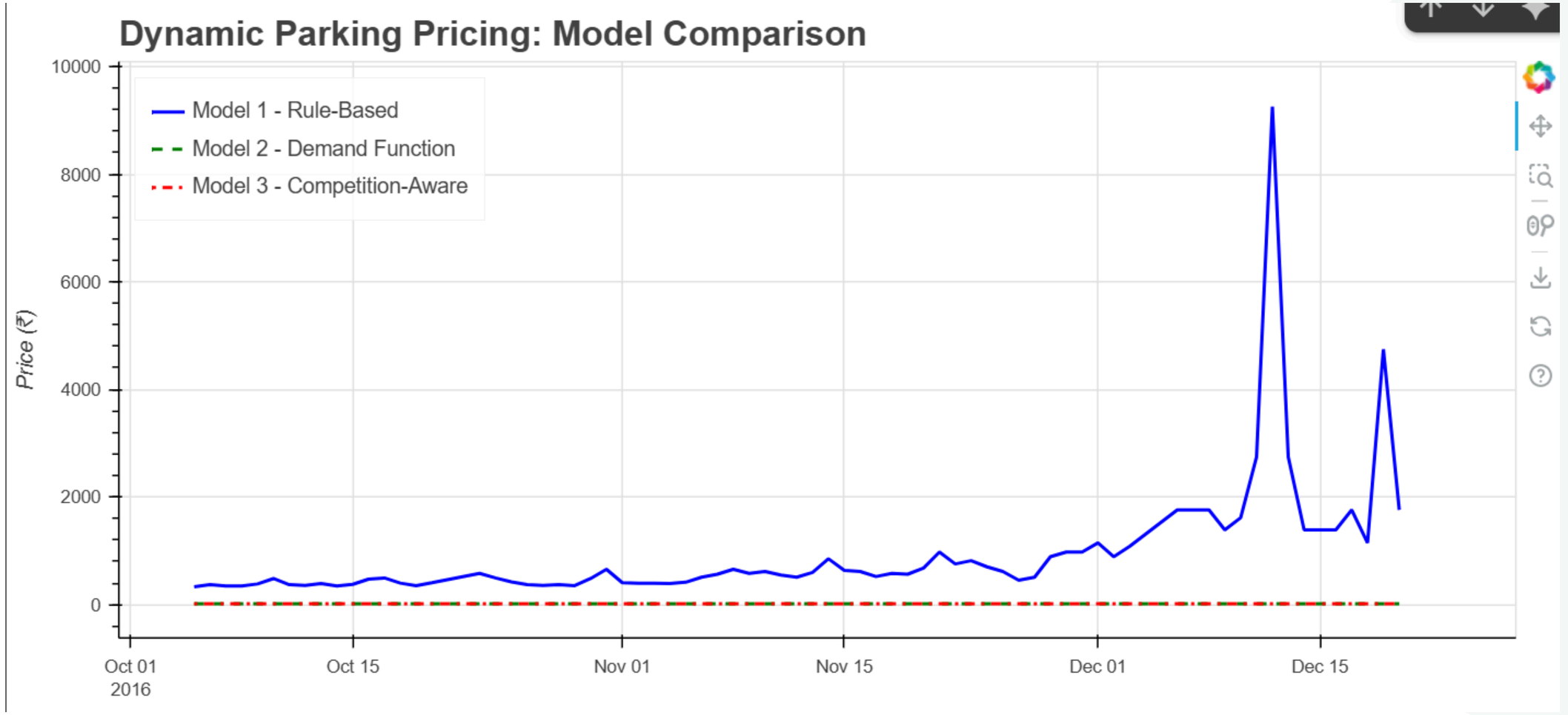
click here to open the output file

```python
# Load CSVs
df1 = pd.read_csv("output_prices1.csv")
df2 = pd.read_csv("output_prices2.csv")
df3 = pd.read_csv("output_prices3.csv")

# Convert timestamp to datetime
df1["t"] = pd.to_datetime(df1["t"])
df2["t"] = pd.to_datetime(df2["t"])
df3["t"] = pd.to_datetime(df3["t"])

# Plot
p = figure(title="Dynamic Parking Pricing: Model Comparison", x_axis_type='datetime', width=950, height=450)

# Add lines
l1 = p.line(df1["t"], df1["price"], color="blue", line_width=2, legend_label="Model 1 - Rule-Based")
l2 = p.line(df2["t"], df2["price"], color="green", line_width=2, line_dash="dashed", legend_label="Model 2 - Demand Function")
l3 = p.line(df3["t"], df3["adjusted_price"], color="red", line_width=2, line_dash="dotdash", legend_label="Model 3 - Competition-Aware")

# Labels and legend
p.xaxis.axis_label = "Timestamp"
p.yaxis.axis_label = "Price (₹)"
p.legend.location = "top_left"
p.legend.click_policy = "hide"
```

- Comparing Model 1, 2, and 3 – Dynamic Pricing Over Time

graph is on next page

Loaded price outputs from all 3 models and plotted them using Bokeh.
Model 1 (Rule-Based): Basic pricing based on occupancy min/max.
Model 2 (Demand-Based): Includes queue, traffic, special days, and vehicle type.
Model 3 (Competition-Aware): Adjusts dynamically based on nearby competitors' pricing.
Interactive Bokeh Plot allows users to compare behavior over time and toggle models.
This visual clearly demonstrates increased sophistication and market reactivity from Model 1 → Model 3.


Dynamic Parking Pricing: Model Comparison
Model 1 - Rule-Based
Model 2 - Demand Function
Model 3 - Competition-Aware
Price (₹)
10000
8000
6000
4000
2000
0
Oct 01
2016
Oct 15
Nov 01
Nov 15
Dec 01
Dec 15

```
output_notebook()

# Load Model 3 output
df = pd.read_csv("output_prices3.csv")
df["t"] = pd.to_datetime(df["t"])
df = df.sort_values("t")

# Create data source with empty initial data
source = ColumnDataSource(data=dict(t=[], price=[], adjusted_price=[], competitor_price=[]))

# Set up figure
p = figure(
    title="Simulated Real-Time Pricing - Model 3",
    x_axis_type="datetime",
    width=900, height=450
)
p.line("t", "price", source=source, line_width=2, color="blue", legend_label="Base Price")
p.line("t", "adjusted_price", source=source, line_width=2, color="green", line_dash="dashed", legend_label="Adjusted Price
p.line("t", "competitor_price", source=source, line_width=2, color="red", line_dash="dotdash", legend_label="Competitor Pr

p.legend.location = "top_left"
p.xaxis.axis_label = "Time"
p.yaxis.axis_label = "Price (₹)"
```

```
# Display plot
handle = show(p, notebook_handle=True)

# Simulate real-time by adding one row at a time
for i in range(len(df)):
    new_data = {
        't': [df.iloc[i]["t"]],
        'price': [df.iloc[i]["price"]],
        'adjusted_price': [df.iloc[i]["adjusted_price"]],
        'competitor_price': [df.iloc[i]["competitor_price"]]
    }
    source.stream(new_data, rollover=100)
    show(p, notebook_handle=True)
    sleep(0.3)  # Pause to simulate streaming
```

- # Model 3: Real-Time Pricing Simulation with Bokeh

Used Bokeh’s streaming tools to simulate live price updates from Model 3.
Plotted:

- Base Price
- Adjusted Price (after competition logic)
- Mean Competitor Price

Streamed one row at a time with a 0.3s delay to mimic real-time updates.
Interactive chart enables tracking how prices respond to demand and competitors over time.
Demonstrates the reactive, real-time behavior of our most advanced pricing model.

it is real time graph, view it in google collab

THANK YOU