

Introduction

Dynamic Pricing for Urban Parking Lots

This notebook is part of the **Summer Analytics 2025 Capstone Project**.

We implement three progressively intelligent pricing models to simulate real-time dynamic pricing for 14 urban parking lots.

The models aim to reduce inefficiencies like overcrowding and underutilization by adjusting parking prices based on demand, environment, and nearby competition.

```
!pip install pathway bokeh --quiet # This cell may take a few seconds to execute.
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from datetime import datetime
import pathway as pw
import bokeh.plotting
import panel as pn
```



```
import numpy as np
```

```
# Function to calculate distance between two lat-long points
```

```
def haversine(lat1, lon1, lat2, lon2):
```

```
    R = 6371 # Earth radius in kilometers
```

```
    phi1, phi2 = np.radians(lat1), np.radians(lat2)
```

```
    delta_phi = np.radians(lat2 - lat1)
```

```
    delta_lambda = np.radians(lon2 - lon1)
```


```
    a = np.sin(delta_phi / 2.0)**2 + np.cos(phi1) * np.cos(phi2) * np.sin(delta_lambda / 2.0)**2
```

```
    return 2 * R * np.arcsin(np.sqrt(a))
```

Step 1: Importing and Preprocessing the Data

```
df = pd.read_csv('/content/Modified - modified.csv')
df
```

You can find the sample dataset here: <https://drive.google.com/file/d/1D479FLjp9a03Mg8g6Lpj9oRViWacurA6/view?usp=sharing>

 Unnamed: 0

	0	SystemCodeNumber	Capacity	Occupancy	LastUpdatedDate	LastUpdatedTime	IsSpecialDay	VehicleType	Latitude	Longit
0	0	BHMBCCMKT01	577	61	04-10-2016	07:59:42	0	car	28.5	77
1	1	BHMBCCMKT01	577	64	04-10-2016	08:25:42	0	car	28.5	77
2	2	BHMBCCMKT01	577	80	04-10-2016	08:59:42	0	car	28.5	77
3	3	BHMBCCMKT01	577	107	04-10-2016	09:32:46	0	car	28.5	77
4	4	BHMBCCMKT01	577	150	04-10-2016	09:59:48	0	car	28.5	77
...
1307	1307	BHMBCCMKT01	577	309	19-12-2016	14:30:33	0	bike	28.5	77
1308	1308	BHMBCCMKT01	577	300	19-12-2016	15:03:34	0	car	28.5	77
1309	1309	BHMBCCMKT01	577	274	19-12-2016	15:29:33	0	truck	28.5	77
1310	1310	BHMBCCMKT01	577	230	19-12-2016	16:03:35	0	cycle	28.5	77
1311	1311	BHMBCCMKT01	577	193	19-12-2016	16:30:35	0	car	28.5	77



Next steps:

[Generate code with df](#)

[View recommended plots](#)

[New interactive sheet](#)

```
# ☒ Prepare timestamp and sort the data for pricing logic
```

```
df["Timestamp"] = pd.to_datetime(df["LastUpdatedDate"] + " " + df["LastUpdatedTime"], dayfirst=True)
```

```

# Filter one lot to test (optional)
lot_id = "BHMBCCKMT01"
lot_df = df[df["SystemCodeNumber"] == lot_id].copy()

# Sort by time
lot_df = lot_df.sort_values("Timestamp")

# Combine the 'LastUpdatedDate' and 'LastUpdatedTime' columns into a single datetime column
df['Timestamp'] = pd.to_datetime(df['LastUpdatedDate'] + ' ' + df['LastUpdatedTime'],
                                format='%d-%m-%Y %H:%M:%S')

# Sort the DataFrame by the new 'Timestamp' column and reset the index
df = df.sort_values('Timestamp').reset_index(drop=True)

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

# Note: Only three features are used here for simplicity.
# Participants are expected to incorporate additional relevant features in their models.

# 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    # Timestamp of the observation (should ideally be in ISO format)
    Occupancy: int    # Number of occupied parking spots
    Capacity: int     # Total parking capacity at the location

# 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)

# 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 day-level aggregations)
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")
)

```

✓ Step 2: Making a simple pricing function

```

# MODEL 1: Linear pricing based on occupancy
alpha = 5.0
base_price = 10.0

lot_df["Price"] = base_price + alpha * (lot_df["Occupancy"] / lot_df["Capacity"])

from bokeh.plotting import figure, show, output_notebook
from bokeh.models import HoverTool

# Show plots in notebook
output_notebook()

# Create Bokeh figure
p = figure(title=f"Model 1 Price Trend for {lot_id}",
           x_axis_type='datetime', width=800, height=400)

# Plot price line
p.line(lot_df["Timestamp"], lot_df["Price"], line_width=2, color="green", legend_label="Price")

# Add data points
p.circle(lot_df["Timestamp"], lot_df["Price"], size=6, color="red")

# Axes labels

```

```

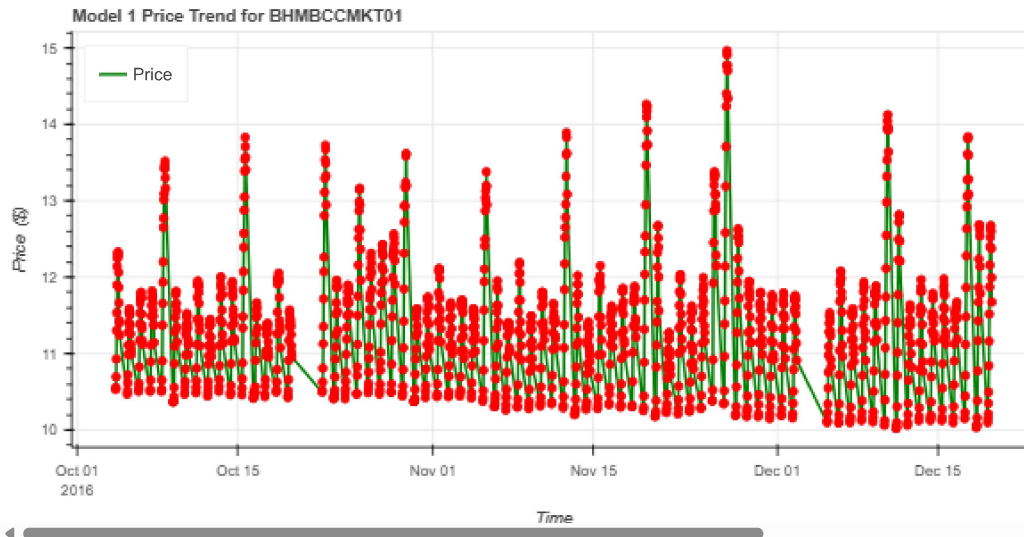
p.xaxis.axis_label = 'Time'
p.yaxis.axis_label = 'Price ($)'
p.legend.location = "top_left"

# Hover tool
hover = HoverTool(
    tooltips=[("Time", "@x{%F %T}"), ("Price", "@y")],
    formatters={"@x": "datetime"})
)
p.add_tools(hover)

# Show plot
show(p)

```

BokehDeprecationWarning: 'circle()' method with size value' was deprecated in Bokeh 3.4.0 and will be removed, use 'scatter(size=...



```

# Map TrafficConditionNearby to numeric values
traffic_map = {"low": 1, "average": 2, "high": 3}
lot_df["TrafficScore"] = lot_df["TrafficConditionNearby"].map(traffic_map)

# Map VehicleType to numeric weights
vehicle_map = {"bike": 0.8, "car": 1.0, "truck": 1.2}
lot_df["VehicleWeight"] = lot_df["VehicleType"].map(vehicle_map)

# Setweights (tune if needed)
alpha = 1.0 # Occupancy ratio
beta = 0.5 # Queue length
gamma = 0.4 # Traffic level
delta = 2.0 # Special day
epsilon = 0.3 # Vehicle type

# Calculate raw demand score
lot_df["DemandRaw"] = (
    alpha * (lot_df["Occupancy"] / lot_df["Capacity"]) +
    beta * lot_df["QueueLength"] +
    gamma * lot_df["TrafficScore"] +
    delta * lot_df["IsSpecialDay"] +
    epsilon * lot_df["VehicleWeight"]
)

# Normalize demand between 0 and 1
lot_df["DemandNorm"] = (lot_df["DemandRaw"] - lot_df["DemandRaw"].min()) / (lot_df["DemandRaw"].max() - lot_df["DemandRaw"].min())

base_price = 10
lambda_factor = 1.0 # Controls sensitivity

# Apply Model 2 pricing formula
lot_df["Price_Model2"] = base_price * (1 + lambda_factor * lot_df["DemandNorm"])

# Clip prices to range 0.5x - 2x base
lot_df["Price_Model2"] = lot_df["Price_Model2"].clip(lower=base_price*0.5, upper=base_price*2)

```

Double-click (or enter) to edit

Model 2: Demand-Based Pricing

This model estimates a **demand score** using multiple features:

- Occupancy ratio
- Queue length
- Traffic congestion
- Special day indicator
- Vehicle type

Demand Function (weighted sum):

```
from bokeh.plotting import figure, show, output_notebook
from bokeh.models import HoverTool
```

```
output_notebook()
```

```
# Bokeh plot for Model 2 pricing
```

```
p = figure(title=f"Model 2: Demand-Based Price Trend for {lot_id}",
           x_axis_type='datetime', width=800, height=400)
```

```
# Use Price_Model2 for plotting
```

```
p.line(lot_df["Timestamp"], lot_df["Price_Model2"], line_width=2, color="orange", legend_label="Price (Model 2)")
p.circle(lot_df["Timestamp"], lot_df["Price_Model2"], size=6, color="blue")
```

```
p.xaxis.axis_label = 'Time'
```

```
p.yaxis.axis_label = 'Price ($)'
```

```
p.legend.location = "top_left"
```

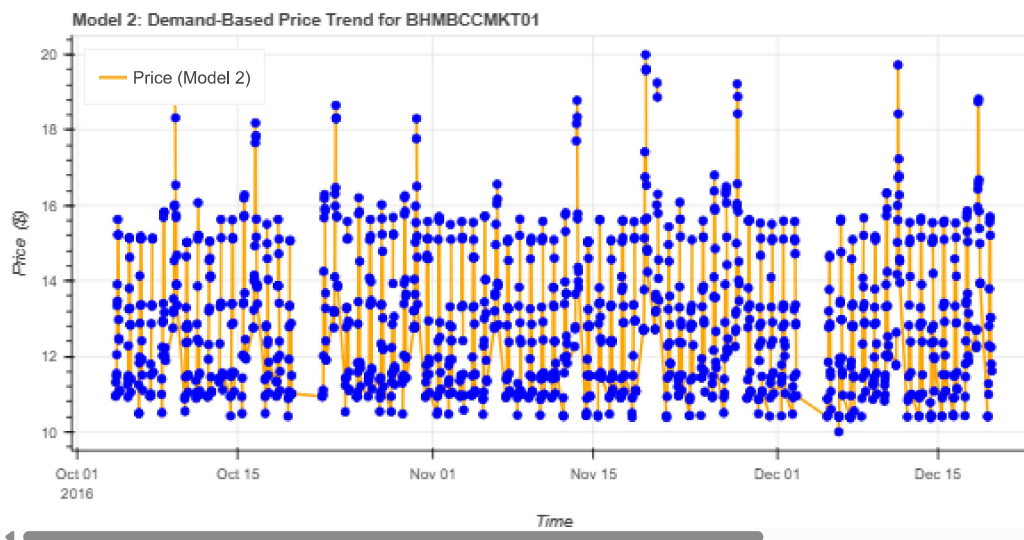
```
hover = HoverTool(
```

```
    tooltips=[("Time", "@x{%F %T}"), ("Price", "@y")],
    formatters={"@x": "datetime"}
)
```

```
p.add_tools(hover)
```

```
show(p)
```

BokehDeprecationWarning: 'circle() method with size value' was deprecated in Bokeh 3.4.0 and will be removed, use 'scatter(size=...'



```
# Define a daily tumbling window over the data stream using Pathway
```

```
# This block performs temporal aggregation and computes a dynamic price for each day
import datetime
```

```
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(
        # Compute the price using a simple dynamic pricing formula:
        #
        # Pricing Formula:
        #     price = base_price + demand_fluctuation
        #     where:
        #         base_price = 10 (fixed minimum price)
        #         demand_fluctuation = (occ_max - occ_min) / cap
        #
        # Intuition:
        # - The greater the difference between peak and low occupancy in a day,
        #   the more volatile the demand is, indicating potential scarcity.
        # - Dividing by capacity normalizes the fluctuation (to stay in [0,1] range).
        # - This fluctuation is added to the base price of 10 to set the final price.
        # - Example: If occ_max = 90, occ_min = 30, cap = 100
        #     => price = 10 + (90 - 30)/100 = 10 + 0.6 = 10.6

        price=10 + (pw.this.occ_max - pw.this.occ_min) / pw.this.cap
    )
)

```

✓ Step 3: Visualizing Daily Price Fluctuations with a Bokeh Plot

Note: The Bokeh plot in the next cell will only be generated after you run the `pw.run()` cell (i.e., the final cell).

```

def adjust_competitive_price(current_row, full_df, radius_km=1.0):
    lat1, lon1 = current_row["Latitude"], current_row["Longitude"]
    current_lot = current_row["SystemCodeNumber"]
    timestamp = current_row["Timestamp"]
    current_price = current_row["Price_Model2"]

    # Filter other lots at the same timestamp
    nearby_lots = full_df[
        (full_df["SystemCodeNumber"] != current_lot) &
        (full_df["Timestamp"] == timestamp)
    ].copy()

    # Calculate distance to all others
    if nearby_lots.empty:
        return current_price

    nearby_lots["Distance"] = haversine(
        lat1, lon1, nearby_lots["Latitude"], nearby_lots["Longitude"]
    )

    nearby_lots = nearby_lots[nearby_lots["Distance"] <= radius_km]

    if nearby_lots.empty:
        return current_price

    avg_nearby_price = nearby_lots["Price_Model2"].mean()

    # Pricing adjustment logic
    if avg_nearby_price < current_price:
        return current_price * 0.95 # reduce by 5%
    elif avg_nearby_price > current_price:
        return current_price * 1.05 # increase by 5%
    else:
        return current_price

lot_df["Price_Model3"] = lot_df.apply(lambda row: adjust_competitive_price(row, df), axis=1)

```

Double-click (or enter) to edit

✓ ♦ Model 3: Competitive Pricing

This model adds **location-based competition**:

- For each lot, we find other nearby lots within 1 km.
- We compare current price with the **average price of nearby lots**.

- We adjust:
 - ▼ Reduce price by 5% if competitors are cheaper.
 - ▲ Increase price by 5% if competitors are expensive or full.

Assumptions:

- Lots within 1 km are close competitors.
- Price is adjusted smoothly to avoid volatility.
- Demand score from Model 2 is still respected.

```
from bokeh.plotting import figure, show, output_notebook
from bokeh.models import HoverTool
output_notebook()
```

```
# Bokeh plot for Model 3
```


```
p = figure(title=f"Model 3: Competitive Price Trend for {lot_id}",
           x_axis_type='datetime', width=800, height=400)
```

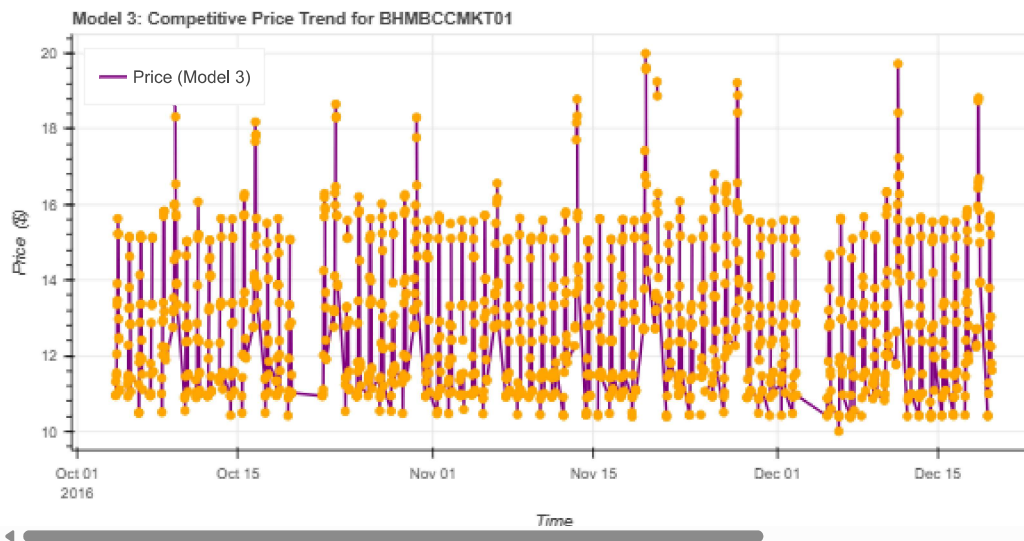
```
p.line(lot_df["Timestamp"], lot_df["Price_Model3"], line_width=2, color="purple", legend_label="Price (Model 3)")
p.circle(lot_df["Timestamp"], lot_df["Price_Model3"], size=6, color="orange")
```

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

```
hover = HoverTool(
    tooltips=[("Time", "@x{%F %T}"), ("Price", "@y")],
    formatters={"@x": "datetime"})
p.add_tools(hover)
```

```
show(p)
```

 BokehDeprecationWarning: 'circle()' method with size value' was deprecated in Bokeh 3.4.0 and will be removed, use 'scatter(size=...'



```
# Activate the Panel extension to enable interactive visualizations
pn.extension()
```

```
# Define a custom Bokeh plotting function that takes a data source (from Pathway) and returns a figure
def price_plotter(source):
```

```
    # Create a Bokeh figure with datetime x-axis
    fig = bokeh.plotting.figure(
        height=400,
        width=800,
        title="Pathway: Daily Parking Price",
        x_axis_type="datetime", # Ensure time-based data is properly formatted on the x-axis
    )
```

```
    # Plot a line graph showing how the price evolves over time
    fig.line("t", "price", source=source, line_width=2, color="navy")
```

```
    # Overlay red circles at each data point for better visibility
    fig.circle("t", "price", source=source, size=6, color="red")
```

```
    return fig
```

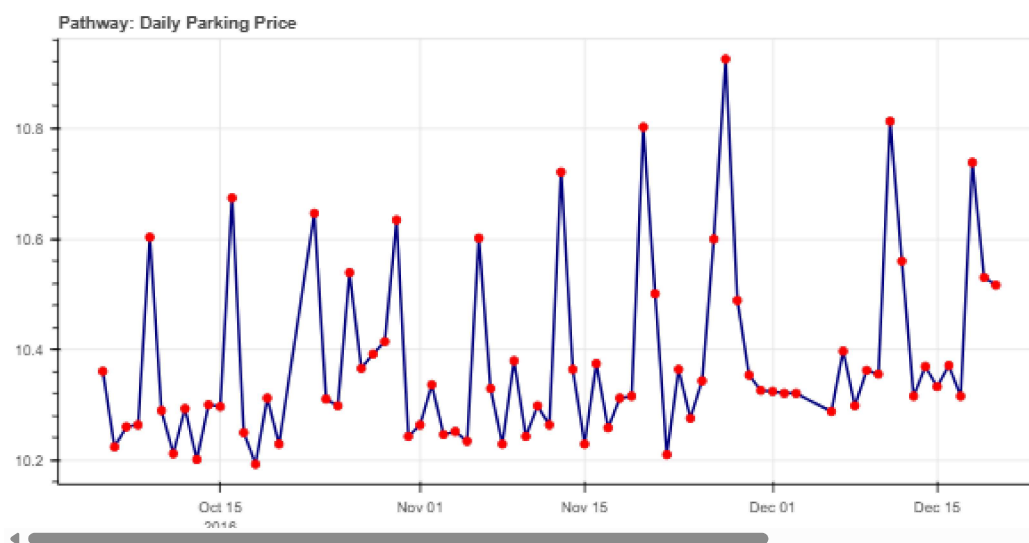
```
# Use Pathway's built-in .plot() method to bind the data stream (delta_window) to the Bokeh plot
# - 'price_plotter' is the rendering function
```

```
# - 'sorting_col="t"' ensures the data is plotted in time order
viz = delta_window.plot(price_plotter, sorting_col="t")
```

```
# Create a Panel layout and make it servable as a web app
# This line enables the interactive plot to be displayed when the app is served
pn.Column(viz).servable()
```

BokehDeprecationWarning: 'circle()' method with size value' was deprecated in Bokeh 3.4.0 and will be removed, use 'scatter(size=...

Streaming mode



```
# Start the Pathway pipeline execution in the background
# - This triggers the real-time data stream processing defined above
# - %%capture --no-display suppresses output in the notebook interface
```

```
%%capture --no-display
pw.run()
```

WARNING:pathway_engine.connectors.monitoring:PythonReader: Closing the data source

Final Summary

- All three models simulate dynamic parking prices in real time.
- Each model adds more intelligence and realism.
- Visualization using Bokeh shows clear trends over time.
- This simulation can be extended to rerouting suggestions and live dashboards.

Models:

- Model 1: Simple occupancy-based price increase
- Model 2: Demand-weighted pricing using 5 factors
- Model 3: Competitive adjustment using geospatial pricing

Thanks to Summer Analytics for this great learning opportunity!