

▸ *Dynamic Pricing Strategy using Python*

Let's start the task of building a dynamic pricing strategy by importing the necessary Python libraries and the dataset:

```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

```
data = pd.read_csv("/content/dynamic_pricing.csv")
print(data.head())
```

	Number_of_Riders	Number_of_Drivers	Location_Category	\
0	90	45	Urban	
1	58	39	Suburban	
2	42	31	Rural	
3	89	28	Rural	
4	78	22	Rural	

	Customer_Loyalty_Status	Number_of_Past_Rides	Average_Ratings	\
0	Silver	13	4.47	
1	Silver	72	4.06	
2	Silver	0	3.99	
3	Regular	67	4.31	
4	Regular	74	3.77	

	Time_of_Booking	Vehicle_Type	Expected_Ride_Duration	\
0	Night	Premium	90	
1	Evening	Economy	43	
2	Afternoon	Premium	76	
3	Afternoon	Premium	134	
4	Afternoon	Economy	149	

	Historical_Cost_of_Ride
0	284.257273
1	173.874753
2	329.795469
3	470.201232
4	579.681422

Exploratory Data Analysis

Let's have a look at the descriptive statistics of the data:

```
print(data.describe())
```

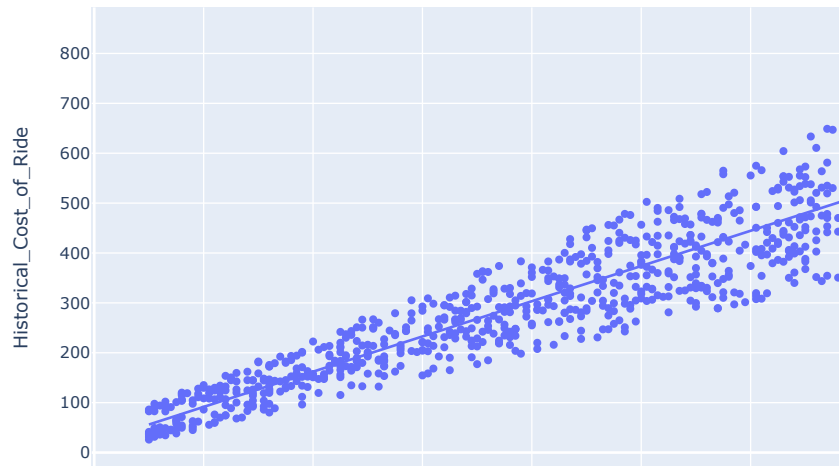
	Number_of_Riders	Number_of_Drivers	Number_of_Past_Rides	\
count	1000.000000	1000.000000	1000.000000	
mean	60.372000	27.076000	50.031000	
std	23.701506	19.068346	29.313774	
min	20.000000	5.000000	0.000000	
25%	40.000000	11.000000	25.000000	
50%	60.000000	22.000000	51.000000	
75%	81.000000	38.000000	75.000000	
max	100.000000	89.000000	100.000000	

	Average_Ratings	Expected_Ride_Duration	Historical_Cost_of_Ride
count	1000.000000	1000.000000	1000.000000
mean	4.257220	99.58800	372.502623
std	0.435781	49.16545	187.158756
min	3.500000	10.00000	25.993449
25%	3.870000	59.75000	221.365202
50%	4.270000	102.00000	362.019426
75%	4.632500	143.00000	510.497504
max	5.000000	180.00000	836.116419

Now let's have a look at the relationship between expected ride duration and the historical cost of the ride:

```
fig = px.scatter(data, x='Expected_Ride_Duration',
                 y='Historical_Cost_of_Ride',
                 title='Expected Ride Duration vs. Historical Cost of Ride',
                 trendline='ols')
fig.show()
```

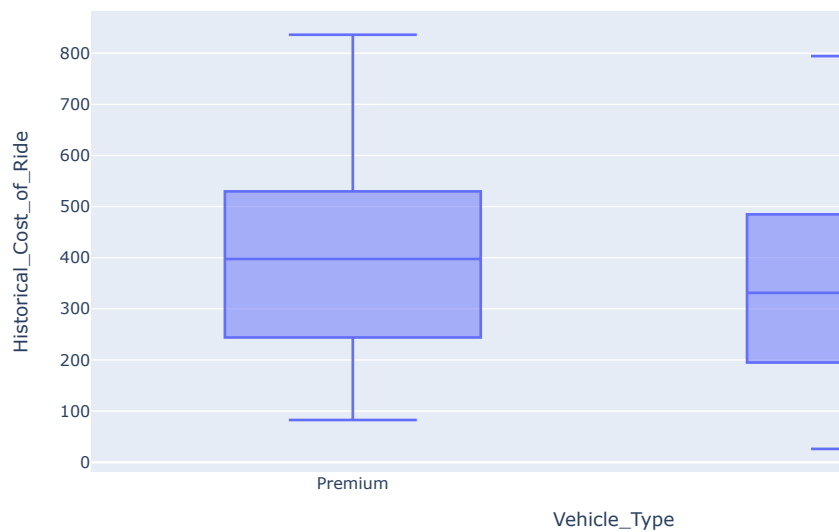
Expected Ride Duration vs. Historical Cost of Ride



Now let's have a look at the distribution of the historical cost of rides based on the vehicle type:

```
fig = px.box(data, x='Vehicle_Type',
             y='Historical_Cost_of_Ride',
             title='Historical Cost of Ride Distribution by Vehicle Type')
fig.show()
```

Historical Cost of Ride Distribution by Vehicle Type



Now let's have a look at the correlation matrix:

```
corr_matrix = data.corr()

fig = go.Figure(data=go.Heatmap(z=corr_matrix.values,
                                x=corr_matrix.columns,
                                y=corr_matrix.columns,
                                colorscale='Viridis'))
fig.update_layout(title='Correlation Matrix')
fig.show()
```

```
<ipython-input-5-7389f30d19e1>:1: FutureWarning:
The default value of numeric_only in DataFrame.corr is deprecated. In a future versi
```

Correlation Matrix



Implementing a Dynamic Pricing Strategy

The data provided by the company states that the company uses a pricing model that only takes the expected ride duration as a factor to determine the price for a ride. Now, we will implement a dynamic pricing strategy aiming to adjust the ride costs dynamically based on the demand and supply levels observed in the data. It will capture high-demand periods and low-supply scenarios to increase prices, while low-demand periods and high-supply situations will lead to price reductions.

Here's how to implement this dynamic pricing strategy using Python:

```
import numpy as np

# Calculate demand_multiplier based on percentile for high and low demand
high_demand_percentile = 75
low_demand_percentile = 25

data['demand_multiplier'] = np.where(data['Number_of_Riders'] > np.percentile(data['Number_of_Riders'], high_demand_percentile),
                                     data['Number_of_Riders'] / np.percentile(data['Number_of_Riders'], high_demand_percentile),
                                     data['Number_of_Riders'] / np.percentile(data['Number_of_Riders'], low_demand_percentile))

# Calculate supply_multiplier based on percentile for high and low supply
high_supply_percentile = 75
low_supply_percentile = 25

data['supply_multiplier'] = np.where(data['Number_of_Drivers'] > np.percentile(data['Number_of_Drivers'], low_supply_percentile),
                                     np.percentile(data['Number_of_Drivers'], high_supply_percentile) / data['Number_of_Drivers'],
                                     np.percentile(data['Number_of_Drivers'], low_supply_percentile) / data['Number_of_Drivers'])

# Define price adjustment factors for high and low demand/supply
demand_threshold_high = 1.2 # Higher demand threshold
demand_threshold_low = 0.8 # Lower demand threshold
supply_threshold_high = 0.8 # Higher supply threshold
supply_threshold_low = 1.2 # Lower supply threshold

# Calculate adjusted_ride_cost for dynamic pricing
data['adjusted_ride_cost'] = data['Historical_Cost_of_Ride'] * (
    np.maximum(data['demand_multiplier'], demand_threshold_low) *
    np.maximum(data['supply_multiplier'], supply_threshold_high)
)
```

In the above code, we first calculated the demand multiplier by comparing the number of riders to percentiles representing high and low demand levels. If the number of riders exceeds the percentile for high demand, the demand multiplier is set as the number of riders divided by the high-demand percentile. Otherwise, if the number of riders falls below the percentile for low demand, the demand multiplier is set as the number of riders divided by the low-demand percentile.

Next, we calculated the supply multiplier by comparing the number of drivers to percentiles representing high and low supply levels. If the number of drivers exceeds the low-supply percentile, the supply multiplier is set as the high-supply percentile divided by the number of drivers. On the other hand, if the number of drivers is below the low-supply percentile, the supply multiplier is set as the low-supply percentile divided by the number of drivers.

Finally, we calculated the adjusted ride cost for dynamic pricing. It multiplies the historical cost of the ride by the maximum of the demand multiplier and a lower threshold (demand_threshold_low), and also by the maximum of the supply multiplier and an upper threshold (supply_threshold_high). This multiplication ensures that the adjusted ride cost captures the combined effect of demand and supply multipliers, with the thresholds serving as caps or floors to control the price adjustments.

Now let's calculate the profit percentage we got after implementing this dynamic pricing strategy:

```
# Calculate the profit percentage for each ride
data['profit_percentage'] = ((data['adjusted_ride_cost'] - data['Historical_Cost_of_Ride']) / data['Historical_Cost_of_Ride']) * 100
# Identify profitable rides where profit percentage is positive
profitable_rides = data[data['profit_percentage'] > 0]

# Identify loss rides where profit percentage is negative
loss_rides = data[data['profit_percentage'] < 0]

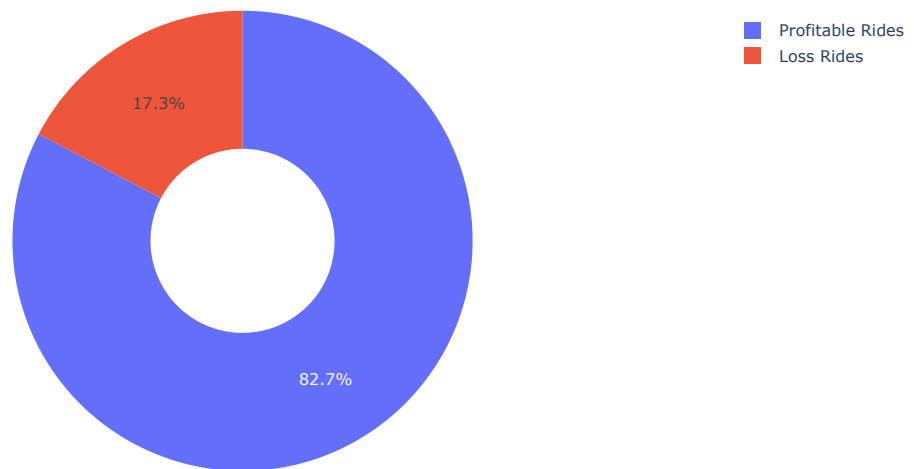
import plotly.graph_objects as go

# Calculate the count of profitable and loss rides
profitable_count = len(profitable_rides)
loss_count = len(loss_rides)

# Create a donut chart to show the distribution of profitable and loss rides
labels = ['Profitable Rides', 'Loss Rides']
values = [profitable_count, loss_count]

fig = go.Figure(data=[go.Pie(labels=labels, values=values, hole=0.4)])
fig.update_layout(title='Profitability of Rides (Dynamic Pricing vs. Historical Pricing)')
fig.show()
```

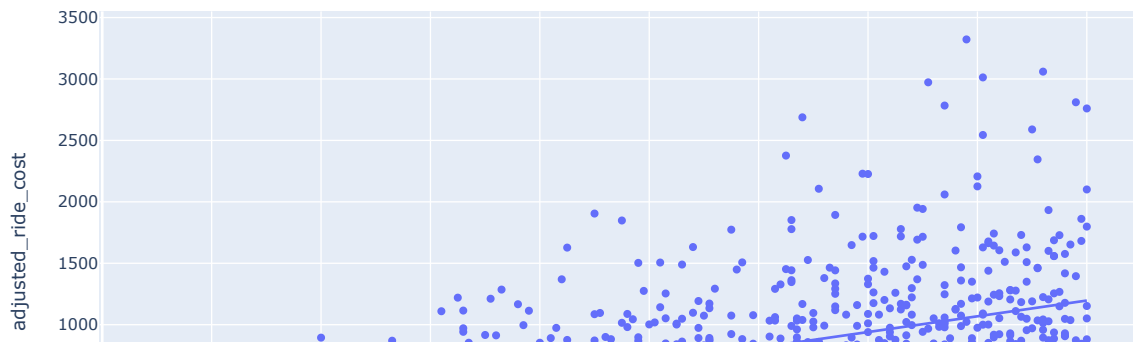
Profitability of Rides (Dynamic Pricing vs. Historical Pricing)



Now let's have a look at the relationship between the expected ride duration and the cost of the ride based on the dynamic pricing strategy:

```
fig = px.scatter(data,
                 x='Expected_Ride_Duration',
                 y='adjusted_ride_cost',
                 title='Expected Ride Duration vs. Cost of Ride',
                 trendline='ols')
fig.show()
```

Expected Ride Duration vs. Cost of Ride



Training a Predictive Model

Now, as we have implemented a dynamic pricing strategy, let's train a Machine Learning model. Before training the model, let's preprocess the data:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

def data_preprocessing_pipeline(data):
    #Identify numeric and categorical features
    numeric_features = data.select_dtypes(include=['float', 'int']).columns
    categorical_features = data.select_dtypes(include=['object']).columns

    #Handle missing values in numeric features
    data[numeric_features] = data[numeric_features].fillna(data[numeric_features].mean())

    #Detect and handle outliers in numeric features using IQR
    for feature in numeric_features:
        Q1 = data[feature].quantile(0.25)
        Q3 = data[feature].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - (1.5 * IQR)
        upper_bound = Q3 + (1.5 * IQR)
        data[feature] = np.where((data[feature] < lower_bound) | (data[feature] > upper_bound),
                                data[feature].mean(), data[feature])

    #Handle missing values in categorical features
    data[categorical_features] = data[categorical_features].fillna(data[categorical_features].mode().iloc[0])

    return data
```

In the above code, we have implemented a data preprocessing pipeline to preprocess the data. You can learn more about it [here](#). As vehicle type is a valuable factor, let's convert it into a numerical feature before moving forward:

```
data["Vehicle_Type"] = data["Vehicle_Type"].map({"Premium": 1,
                                                  "Economy": 0})
```

Now let's split the data and train a Machine Learning model to predict the cost of a ride:

```
#splitting data
from sklearn.model_selection import train_test_split
x = np.array(data[["Number_of_Riders", "Number_of_Drivers", "Vehicle_Type", "Expected_Ride_Duration"]])
y = np.array(data[["adjusted_ride_cost"]])

x_train, x_test, y_train, y_test = train_test_split(x,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)

# Reshape y to 1D array
y_train = y_train.ravel()
y_test = y_test.ravel()

# Training a random forest regression model
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
model.fit(x_train, y_train)
```

```
▼ RandomForestRegressor
RandomForestRegressor()
```

Now let's test this Machine Learning model using some input values:

```
def get_vehicle_type_numeric(vehicle_type):
    vehicle_type_mapping = {
        "Premium": 1,
        "Economy": 0
    }
    vehicle_type_numeric = vehicle_type_mapping.get(vehicle_type)
    return vehicle_type_numeric

# Predicting using user input values
def predict_price(number_of_riders, number_of_drivers, vehicle_type, Expected_Ride_Duration):
    vehicle_type_numeric = get_vehicle_type_numeric(vehicle_type)
    if vehicle_type_numeric is None:
        raise ValueError("Invalid vehicle type")

    input_data = np.array([[number_of_riders, number_of_drivers, vehicle_type_numeric, Expected_Ride_Duration]])
    predicted_price = model.predict(input_data)
    return predicted_price

# Example prediction using user input values
user_number_of_riders = 50
user_number_of_drivers = 25
user_vehicle_type = "Economy"
Expected_Ride_Duration = 30
predicted_price = predict_price(user_number_of_riders, user_number_of_drivers, user_vehicle_type, Expected_Ride_Duration)
print("Predicted price:", predicted_price)

Predicted price: [232.26761]
```

Here's a comparison of the actual and predicted results:

```
import plotly.graph_objects as go

# Predict on the test set
y_pred = model.predict(x_test)

# Create a scatter plot with actual vs predicted values
fig = go.Figure()

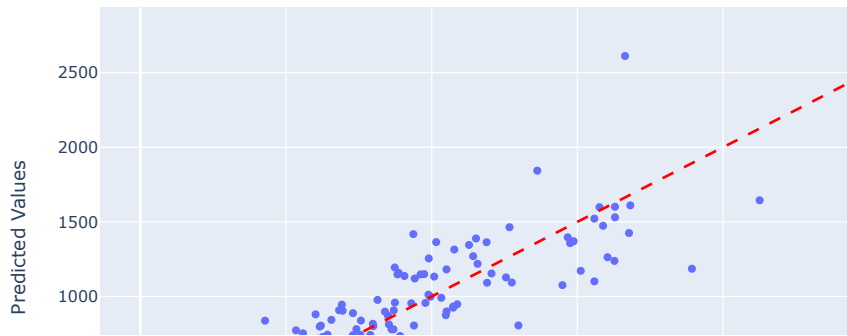
fig.add_trace(go.Scatter(
    x=y_test.flatten(),
    y=y_pred,
    mode='markers',
    name='Actual vs Predicted'
))

# Add a line representing the ideal case
fig.add_trace(go.Scatter(
    x=[min(y_test.flatten()), max(y_test.flatten())],
    y=[min(y_test.flatten()), max(y_test.flatten())],
    mode='lines',
    name='Ideal',
    line=dict(color='red', dash='dash')
))

fig.update_layout(
    title='Actual vs Predicted Values',
    xaxis_title='Actual Values',
    yaxis_title='Predicted Values',
    showlegend=True,
)

fig.show()
```

Actual vs Predicted Values



So this is how you can use Machine Learning to implement a data-driven dynamic pricing strategy using Python.



Summary In a dynamic pricing strategy, the aim is to maximize revenue and profitability by pricing items at the right level that balances supply and demand dynamics. It allows businesses to adjust prices dynamically based on factors like time of day, day of the week, customer segments, inventory levels, seasonal fluctuations, competitor pricing, and market conditions. I hope you liked this article on Dynamic Pricing Strategy using Python.

