Case Study: Dynamic Pricing. Will utilize machine learning (regression) for dynamic pricing. Will analyze company X data on the last 1000 rides and develop a dynamic pricing strategy that can be implemented.

In [29]:
```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
```

In [2]:
```
df = pd.read_csv('dynamic_pricing.csv')
df.head(5)
```

Out[2]:

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

In [3]:
```
print(df.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              1000.00000              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  
```

In [8]:
```
fig = px.scatter(df, x='Expected_Ride_Duration', y='Historical_Cost_of_Ride'
fig.show()
```

In [13]:
```
corr_matrix = df.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()
```

```
/var/folders/vv/3nnd1g4506z6vdqnf44fkr2c0000gn/T/ipykernel_25009/3960197085.
py:1: FutureWarning:

The default value of numeric_only in DataFrame.corr is deprecated. In a futu
re version, it will default to False. Select only valid columns or specify t
he value of numeric_only to silence this warning.
```

After a brief analysis we can see that company X only takes expected ride duration as a determining factor. Let's implement a strategy that will adjust ride costs dynamically based on the demand and supply levels.

In [19]:
```
# get demand and supply multipliers based on percentile for high and low dem
high_percentile = 75
low_percentile = 25
df['demand_multiplier'] = np.where(df['Number_of_Riders'] > np.percentile(df
                                   df['Number_of_Riders'] / np.percentile(df
                                   df['Number_of_Riders'] / np.percentile(df
```

In [20]:
```
df['supply_multiplier'] = np.where(df['Number_of_Drivers'] > np.percentile(c
                                     np.percentile(df['Number_of_Drivers'],
                                     np.percentile(df['Number_of_Drivers'],
```

In [21]:
```
demand_threshold_high = 1.2
demand_threshold_low = 0.8
supply_threshold_high = 0.8
supply_threshold_low = 1.2
```

In [22]:
```
df['adjusted_ride_cost'] = df['Historical_Cost_of_Ride'] * (
    np.maximum(df['demand_multiplier'], demand_threshold_low) *
```

```
    np.maximum(df['supply_multiplier'], supply_threshold_high)
)
```

In [24]:
```
# get potential profit percentage
df['profit_percentage'] = ((df['adjusted_ride_cost'] - df['Historical_Cost_c
# identify profitable rides
profitable_rides = df[df['profit_percentage'] > 0]
# identify losses
loss_rides = df[df['profit_percentage'] < 0]
profitable_count = len(profitable_rides)
loss_count = len(loss_rides)
# visualization
labels = ['Profitable Rides', 'Loss Rides']
values = [profitable_count, loss_count]
fig = go.Figure(data=[go.Pie(labels=labels, values=values, hole=0.2)])
fig.update_layout(title='Profitability of Rides (Dynamic Pricing vs. Histori
fig.show()
```

In [25]:
```
fig = px.scatter(df,
                 x='Expected_Ride_Duration',
                 y='adjusted_ride_cost',
                 title='Expected Ride Diration vs Cost',
```

```
                 trendline='ols')
fig.show()
```

Now let's train ML model

In [27]:

```
# preprocess
def data_preprocessing_pipeline(data):
    # get features
    numeric_features = data.select_dtypes(include=['float', 'int']).columns
    categorical_features = data.select_dtypes(include=['object']).columns
    # handle missing values
    data[numeric_features] = data[numeric_features].fillna(data[numeric_feat
    # handle outliers 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[featu
                                 data[feature].mean(), data[feature])
    # handle missing values
    data[categorical_features] = data[categorical_features].fillna(data[cate
```

```
    return data
```

In [28]:
```
df["Vehicle_Type"] = df["Vehicle_Type"].map({"Premium": 1,
                                           "Economy": 0})
```

In [30]:
```
x = np.array(df[["Number_of_Riders", "Number_of_Drivers", "Vehicle_Type", "E
y = np.array(df[["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()
```

In [31]:
```
model = RandomForestRegressor()
model.fit(x_train, y_train)
```

Out[31]:
```
▾ RandomForestRegressor
RandomForestRegressor()
```

In [32]:
```
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, Expecte
    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_typ
    predicted_price = model.predict(input_data)
    return predicted_price
```

In [33]:
```
# 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_driver
print("Predicted price:", predicted_price)
```

```
Predicted price: [264.21756712]
```

In [34]:
```
# plot actual vs predicted to demonstrate
# predict on test set
y_pred = model.predict(x_test)
```

```
# create a scatter plot
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()
```

```
In [ ]:
```