In [2]:
```
import pandas as pd

# Load the CSV file into a pandas DataFrame

file_path = "/Volumes/Untitled/aca2eb7d00ea1a7b8ebd4e68314663af.csv"
df = pd.read_csv(file_path)
```

In [6]:
```
# Nelder-Mead Algorithm for Pricing Strategy Optimization System

# Igor Mol <igor.mol@makes.ai>

# In this solution to the price optimization problem, a linear regression mo
# is employed to understand the relationship between various features, such
# order item id and product category, and the target variable, typically the
# shipping limit date. This model helps in capturing the underlying patterns
# and associations within the data. To optimize pricing, a Nelder-Mead
# optimization algorithm is then utilized. The Nelder-Mead algorithm
# iteratively adjusts the price parameter, seeking to minimize the negative
# R-squared value obtained from the linear regression model. By doing so, it
# aims to find an optimal price point that maximizes the goodness-of-fit,
# effectively fine-tuning the pricing strategy based on the observed
# relationships between input features and the target variable. This combine
# approach allows for data-driven and iterative adjustments to pricing
# strategies, helping businesses make informed decisions for better overall
# performance.


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
from mpl_toolkits.mplot3d import Axes3D

# Prepare data for our machine learning model.
# Parameters:
#   - df: a Pandas DataFrame containing necessary columns.
# Returns:
#   - X: a DataFrame with selected features (order_item_id, price,
#        product_category_name_english).
#   - y: a Series with the month extracted from the 'shipping_limit_date'
#        column converted to datetime format.

def prepare_data(df):
    # Selecting relevant features for input (X).
    X = df[['order_item_id', 'price', 'product_category_name_english']]

    # Extracting month from 'shipping_limit_date' and converting to datetime
    y = pd.to_datetime(df['shipping_limit_date']).dt.month

    # Returning the prepared features (X) and target variable (y).
    return X, y

# Customize datasets into train-test subsets.
# Parameters:
#   - X: Input features.
#   - y: Target variable.
#   - test_size: Proportion of the dataset to include in the test split.
#   - random_state: Seed for reproducibility.
# Returns:
#   - X_train: Training set features.
#   - X_test: Testing set features.
#   - y_train: Training set target variable.
#   - y_test: Testing set target variable.

def custom_train_test_split(X, y, test_size=0.2, random_state=None):
```

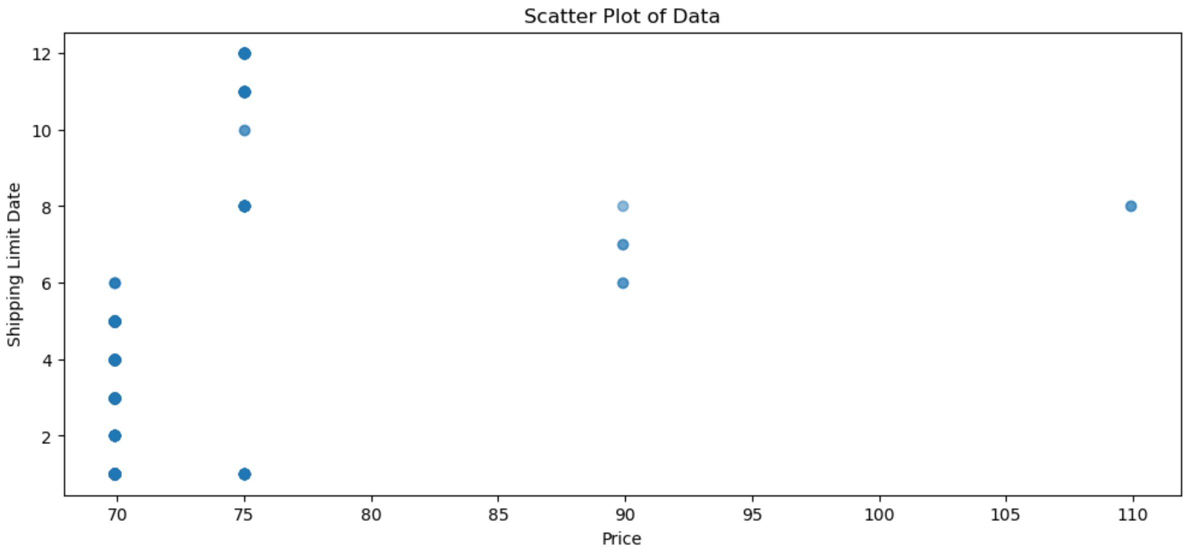
```
    # Setting random seed for reproducibility if random_state is provided.
    if random_state is not None:
        np.random.seed(random_state)

    # Total number of samples in the dataset.
    num_samples = len(X)

    # Creating an array of indices and shuffling it.
    indices = np.arange(num_samples)
    np.random.shuffle(indices)

    # Calculating the number of samples for the test set.
    test_samples = int(test_size * num_samples)

    # Splitting the data into training and testing sets using shuffled indic
    X_train = X.iloc[indices[test_samples:]]
    X_test = X.iloc[indices[:test_samples]]
    y_train = y.iloc[indices[test_samples:]]
    y_test = y.iloc[indices[:test_samples]]

    # Returning the split datasets.
    return X_train, X_test, y_train, y_test

# Train a linear regression model with regularization.
# Parameters:
#   - X_train: Training set features.
#   - y_train: Training set target variable.
#   - alpha: Regularization strength (default is 1e-6).
# Returns:
#   - theta: Coefficients of the linear regression model.

def train_linear_regression(X_train, y_train, alpha=1e-6):
    # Add a column of ones for the bias term to the input features.
    X_train = np.c_[np.ones(X_train.shape[0]), X_train]

    # Create an identity matrix for regularization.
    identity_matrix = np.eye(X_train.shape[1])

    # Compute the coefficients using the closed-form solution.
    theta = inv(X_train.T @ X_train + alpha * identity_matrix) @ X_train.T @

    # Return the trained coefficients.
    return theta

# Train a linear regression model with L2 regularization (ridge regression).
# Parameters:
#   - X_train: Training set features.
#   - y_train: Training set target variable.
#   - alpha: Regularization strength (default is 1e-6).
# Returns:
#   - theta: Coefficients of the linear regression model.
def train_linear_regression(X_train, y_train, alpha=1e-6):
    # Add a column of ones for the bias term to the input features.
    X_train = np.c_[np.ones(X_train.shape[0]), X_train]

    # Create an identity matrix for regularization.
    identity_matrix = np.eye(X_train.shape[1])

    # Compute the coefficients using the closed-form solution with ridge reg
    theta = inv(X_train.T @ X_train + alpha * identity_matrix) @ X_train.T @
```

```
    theta = inv(X_train.T @ X_train + alpha * identity_matrix) @ X_train.T @

    # Return the trained coefficients.
    return theta

# Make predictions using a linear regression model.
# Parameters:
#   - X: Input features for prediction.
#   - theta: Coefficients of the linear regression model.
# Returns:
#   - Predictions based on the input features and model coefficients.
def predict(X, theta):
    # Add a column of ones for the bias term to the input features.
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate predictions using the linear regression model.
    predictions = X @ theta

    # Return the predictions.
    return predictions

# Objective function for optimization, aiming to minimize negative R-squared
# Parameters:
#   - params: Optimization parameters, in this case, a single value represen
#   - X_test: Testing set features.
#   - theta: Coefficients of the linear regression model.
#   - y_test: Testing set target variable.
# Returns:
#   - Negative R-squared value to be minimized.
def objective_function(params, X_test, theta, y_test):
    # Extracting the price parameter from the optimization parameters.
    price = params[0]

    # Creating a copy of X_test and adding the 'price' column.
    X_copy = X_test.copy()
    X_copy['price'] = price

    # Add a column of ones for the bias term to the input features.
    X_copy = np.c_[np.ones(X_copy.shape[0]), X_copy]

    # Calculate residuals and R-squared.
    residuals = y_test - X_copy @ theta
    r_squared = 1 - (np.sum(residuals**2) / np.sum((y_test - np.mean(y_test)

    # Return the negative R-squared value (to be minimized).
    return -r_squared

# Nelder-Mead optimization algorithm for univariate functions.
# Parameters:
#   - func: Objective function to be minimized.
#   - x0: Initial guess for the minimum.
#   - args: Additional arguments to be passed to the objective function.
#   - tol: Tolerance for convergence (default is 1e-6).
#   - max_iter: Maximum number of iterations (default is 1000).
# Returns:
#   - x: Estimated minimum of the objective function.
def minimize_nelder_mead(func, x0, args=(), tol=1e-6, max_iter=1000):
    x = x0.copy()
    for _ in range(max_iter):
        step = np.maximum(np.abs(x) * tol, tol)
```

```
        step = np.maximum(np.abs(x) * tol, tol)
        grad = (func(x + step, *args) - func(x - step, *args)) / (2 * step)
        x = x - step * grad
    return x

# Next function optimizes the pricing strategy using a Nelder-Mead algorithm
# Parameters:
#   - X_test: Testing set features for optimization.
#   - theta: Coefficients of the linear regression model.
#   - y_test: Testing set target variable.
# Returns:
#   - optimal_price: Optimal price obtained from the optimization process.
def optimize_price(X_test, theta, y_test):
    # Initialize the price parameter with the mean of 'price' in the test se
    initial_price = np.mean(X_test['price'])

    # Create a list of initial parameters containing the initial price.
    initial_params = [initial_price]

    # Use the minimize_nelder_mead function to find the optimal price.
    result = minimize_nelder_mead(objective_function, initial_params, args=(

    # Extract the optimal price from the result.
    optimal_price = result[0]

    # Return the optimal price.
    return optimal_price

# Plot a scatter plot of the data.
# Parameters:
#   - X: Input features, assuming a DataFrame with a 'price' column.
#   - y: Target variable, presumably the 'Shipping Limit Date'.
def plot_scatter(X, y):
    # Scatter plot with transparency (alpha) set to 0.5 for better visibilit
    plt.scatter(X['price'], y, alpha=0.5)

    # Setting plot title and axis labels.
    plt.title('Scatter Plot of Data')
    plt.xlabel('Price')
    plt.ylabel('Shipping Limit Date')

    # Show the plot.
    plt.show()

def main():
    # Assuming df is your DataFrame
    X, y = prepare_data(df)
    X_train, X_test, y_train, y_test = custom_train_test_split(X, y, test_si

    # Train linear regression
    theta = train_linear_regression(X_train, y_train)

    # Optimize price
    optimal_price = optimize_price(X_test, theta, y_test)
    print("Optimal Price:", optimal_price)

    # Plotting
    plt.figure(figsize=(12, 5))
    plot_scatter(X, y)
```

```
    # You may want to add a line for the linear regression fit, but it's not

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

```
Optimal Price: 71.88497231597665
```




```
<Figure size 640x480 with 0 Axes>
```

In [ ]: