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):
```

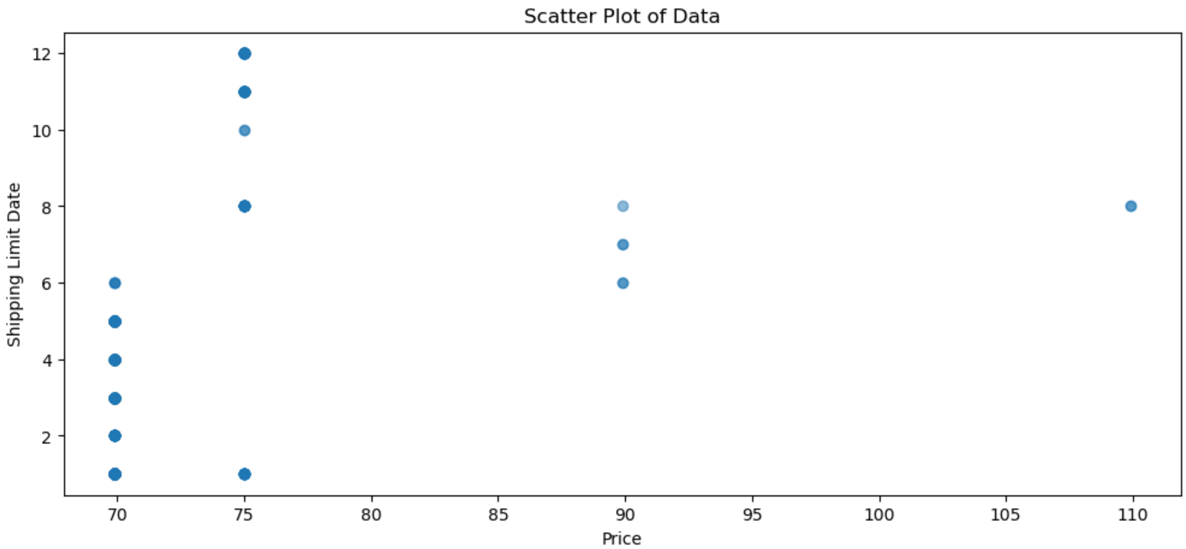
```
    # Setting random seed for reproducibility if random_state is provided.
    if random_state is not None:
        np.random.seed(random_state)

    # 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-squarec
# Parameters:
#   - params: Optimization parameters, in this case, a single value represer
#   - 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 [8]:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from scipy.optimize import minimize

def extract_features_and_target(df):
    X = df[['order_item_id', 'price', 'product_category_name_english']]
    y = pd.to_datetime(df['shipping_limit_date']).dt.month
    return X, y

def split_data(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    return X_train, X_test, y_train, y_test

def objective_function(params, X_train, X_test, y_train, y_test):
    price = params[0]
    X_train['price'] = price
    model = RandomForestRegressor(random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    r_squared = -model.score(X_test, y_test)
    return r_squared

def run_optimization(X_train, X_test, y_train, y_test):
    initial_guess = [X_train['price'].mean()]
    result = minimize(objective_function, initial_guess, args=(X_train.copy(
    optimal_price = result.x[0]
    return optimal_price

def main():
    # Assuming df is your DataFrame
    X, y = extract_features_and_target(df)
    X_train, X_test, y_train, y_test = split_data(X, y)
    optimal_price = run_optimization(X_train, X_test, y_train, y_test)

    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 71.22990654205606
```

In [11]:
```
import pandas as pd
import numpy as np
from scipy.optimize import minimize
from sklearn.tree import DecisionTreeRegressor

class RandomForest:
    def __init__(self, n_estimators=10, random_state=None):
        self.n_estimators = n_estimators
        self.random_state = random_state
        self.estimators = []

    def fit(self, X, y):
        for _ in range(self.n_estimators):
            # Randomly select samples with replacement
            indices = np.random.choice(len(X), size=len(X), replace=True)
            X_sampled = X.iloc[indices]
            y_sampled = y.iloc[indices]

            # Train a decision tree on the sampled data
            tree = DecisionTreeRegressor(random_state=self.random_state)
            tree.fit(X_sampled, y_sampled)

            # Append the trained tree to the list of estimators
            self.estimators.append(tree)

    def predict(self, X):
        # Predict using each tree and average the results
        predictions = np.zeros(len(X))
        for tree in self.estimators:
            predictions += tree.predict(X)
        return predictions / len(self.estimators)

def extract_features_and_target(df):
    X = df[['order_item_id', 'price', 'product_category_name_english']]
    y = pd.to_datetime(df['shipping_limit_date']).dt.month
    return X, y


def split_data(X, y, test_size=0.2, random_state=None):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test
    return X_train, X_test, y_train, y_test

def objective_function(params, X_train, X_test, y_train, y_test):
    price = params[0]
    X_train['price'] = price

    # Train a Random Forest model
    model = RandomForest(n_estimators=10, random_state=42)
    model.fit(X_train, y_train)

    # Predict the target variable on the test data
    y_pred = model.predict(X_test)

    # Calculate the negative R-squared value (minimize negative R-squared)
    residual_sum_of_squares = ((y_test - y_pred) ** 2).sum()
    total_sum_of_squares = ((y_test - y_test.mean()) ** 2).sum()
    r_squared = 1 - (residual_sum_of_squares / total_sum_of_squares)

    return -r_squared
```

```
    return -r_squared

def run_optimization(X_train, X_test, y_train, y_test):
    initial_guess = [X_train['price'].mean()]
    result = minimize(objective_function, initial_guess, args=(X_train.copy(
    optimal_price = result.x[0]
    return optimal_price

def main():
    # Replace the ellipsis with the actual DataFrame initialization
    X, y = extract_features_and_target(df)
    X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, rando
    optimal_price = run_optimization(X_train, X_test, y_train, y_test)

    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 64.99728971960025
```

In [12]:
```
import pandas as pd
import numpy as np
from scipy.optimize import minimize
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

def extract_features_and_target(df):
    X = df[['order_item_id', 'price', 'product_category_name_english']]
    y = pd.to_datetime(df['shipping_limit_date']).dt.month
    return X, y

def split_data(X, y, test_size=0.2, random_state=None):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test
    return X_train, X_test, y_train, y_test

def train_random_forest(X, y, n_estimators=10, random_state=None):
    estimators = []
    for _ in range(n_estimators):
        indices = np.random.choice(len(X), size=len(X), replace=True)
        X_sampled = X.iloc[indices]
        y_sampled = y.iloc[indices]

        tree = DecisionTreeRegressor(random_state=random_state)
        tree.fit(X_sampled, y_sampled)

        estimators.append(tree)

    return estimators

def predict_random_forest(X, estimators):
    predictions = np.zeros(len(X))
    for tree in estimators:
        predictions += tree.predict(X)
    return predictions / len(estimators)

def objective_function(params, X_train, X_test, y_train, y_test):
    price = params[0]
    X_train['price'] = price

    model = train_random_forest(X_train, y_train, n_estimators=10, random_st
    y_pred = predict_random_forest(X_test, model)

    residual_sum_of_squares = ((y_test - y_pred) ** 2).sum()
    total_sum_of_squares = ((y_test - y_test.mean()) ** 2).sum()
    r_squared = 1 - (residual_sum_of_squares / total_sum_of_squares)

    return -r_squared

def run_optimization(X_train, X_test, y_train, y_test):
    initial_guess = [X_train['price'].mean()]
    result = minimize(objective_function, initial_guess, args=(X_train.copy(
    optimal_price = result.x[0]
    return optimal_price

def main():
    # Replace the ellipsis with the actual DataFrame initialization
    X, y = extract_features_and_target(df)
    X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, rando
```

```
    optimal_price = run_optimization(X_train, X_test, y_train, y_test)

    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 68.56223589745636
```

In [13]:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

def extract_features_and_target(df):
    X = df[['order_item_id', 'price', 'product_category_name_english']]
    y = pd.to_datetime(df['shipping_limit_date']).dt.month
    return X, y

def split_data(X, y, test_size=0.2, random_state=None):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test
    return X_train, X_test, y_train, y_test

def train_random_forest(X, y, n_estimators=10, random_state=None):
    estimators = []
    for _ in range(n_estimators):
        indices = np.random.choice(len(X), size=len(X), replace=True)
        X_sampled = X.iloc[indices]
        y_sampled = y.iloc[indices]

        tree = DecisionTreeRegressor(random_state=random_state)
        tree.fit(X_sampled, y_sampled)

        estimators.append(tree)

    return estimators

def predict_random_forest(X, estimators):
    predictions = np.zeros(len(X))
    for tree in estimators:
        predictions += tree.predict(X)
    return predictions / len(estimators)

def calculate_r_squared(y_test, y_pred):
    residual_sum_of_squares = ((y_test - y_pred) ** 2).sum()
    total_sum_of_squares = ((y_test - y_test.mean()) ** 2).sum()
    return 1 - (residual_sum_of_squares / total_sum_of_squares)

def run_optimization(X_train, X_test, y_train, y_test):
    # Perform a grid search for the optimal price
    prices_to_try = np.linspace(X_train['price'].min(), X_train['price'].max

    best_r_squared = -float('inf')
    optimal_price = None

    for price in prices_to_try:
        X_train_copy = X_train.copy()
        X_train_copy['price'] = price

        model = train_random_forest(X_train_copy, y_train, n_estimators=10,
        y_pred = predict_random_forest(X_test, model)

        r_squared = calculate_r_squared(y_test, y_pred)

        if r_squared > best_r_squared:
            best_r_squared = r_squared
            optimal_price = price
```

```
    return optimal_price

def main():
    # Replace the ellipsis with the actual DataFrame initialization
    X, y = extract_features_and_target(df)
    X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, rando
    optimal_price = run_optimization(X_train, X_test, y_train, y_test)

    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 80.40505050505051
```

In [16]:
```
# Random Forests and Decision Trees in Pricing Optimization

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

# In pricing optimization, Random Forests and Decision Trees are utilized to
# model the relationship between product prices and relevant features.
# Decision Trees serve as the basic building blocks, capturing the
# hierarchical decision-making process based on input features. Random
# Forests, on the other hand, utilize an ensemble of Decision Trees to
# enhance predictive accuracy and robustness. For instance, in the
# 'objective_function' here, a Random Forest is trained with various
# 'price' values to maximize the R-squared metric, providing an optimized
# price parameter for pricing strategies, considering the interplay
# between product attributes and pricing.

import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeRegressor

# extract_features_and_target:
# This function extracts features and target variable from a DataFrame.
# Parameters:
#   - df: DataFrame containing the data
# Returns:
#   - X: DataFrame with features (order_item_id, price, product_category_nam
#   - y: Series representing the target variable (month from shipping_limit_

def extract_features_and_target(df):
    # Extracting features (order_item_id, price, product_category_name_engli
    X = df[['order_item_id', 'price', 'product_category_name_english']]

    # Extracting target variable by converting shipping_limit_date to month
    y = pd.to_datetime(df['shipping_limit_date']).dt.month

    # Returning features and target
    return X, y

# training_testing_subsets:
# This function splits input features and target variable into training and
# ting subsets.
# Parameters:
#   - X: DataFrame of features
#   - y: Series of target variable
#   - test_size: Proportion of data to be used for testing (default is 0.2)
#   - random_state: Seed for reproducibility (default is None)
# Returns:
#   - X_train: DataFrame of training features
#   - X_test: DataFrame of testing features
#   - y_train: Series of training target variable
#   - y_test: Series of testing target variable

def training_testing_subsets(X, y, test_size=0.2, random_state=None):
    # Setting seed for reproducibility
    np.random.seed(random_state)

    # Shuffle indices
    indices = np.arange(len(X))
    np.random.shuffle(indices)
```

```
    # Calculate the number of samples for testing
    test_samples = int(test_size * len(X))

    # Split indices into training and testing sets
    train_indices = indices[test_samples:]
    test_indices = indices[:test_samples]

    # Create training and testing subsets
    X_train, X_test = X.iloc[train_indices], X.iloc[test_indices]
    y_train, y_test = y.iloc[train_indices], y.iloc[test_indices]

    # Return training and testing subsets
    return X_train, X_test, y_train, y_test

# train_random_forest:
# This function trains a Random Forest ensemble on the input features and ta
# variable.
# Parameters:
#   - X: DataFrame of features
#   - y: Series of target variable
#   - n_estimators: Number of decision trees in the ensemble (default is 10)
#   - random_state: Seed for reproducibility (default is None)
# Returns:
#   - estimators: List of trained decision trees in the Random Forest ensemb

def train_random_forest(X, y, n_estimators=10, random_state=None):
    # List to store the trained decision trees
    estimators = []

    # Loop to train each decision tree in the ensemble
    for _ in range(n_estimators):
        # Randomly sample with replacement from the data
        indices = np.random.choice(len(X), size=len(X), replace=True)
        X_sampled = X.iloc[indices]
        y_sampled = y.iloc[indices]

        # Create and train a decision tree
        tree = DecisionTreeRegressor(random_state=random_state)
        tree.fit(X_sampled, y_sampled)

        # Append the trained decision tree to the list
        estimators.append(tree)

    # Return the list of trained decision trees
    return estimators

# predict_random_forest:
# This function predicts the target variable using a Random Forest ensemble.
# Parameters:
#   - X: DataFrame of features for prediction
#   - estimators: List of trained decision trees in the Random Forest ensemb
# Returns:
#   - predictions: Array of predicted values for the target variable

def predict_random_forest(X, estimators):
    # Initializing an array to store predictions
    predictions = np.zeros(len(X))
```

```
    # Accumulating predictions from each decision tree in the ensemble
    for tree in estimators:
        predictions += tree.predict(X)

    # Calculating the average prediction across all trees
    return predictions / len(estimators)

# objective_function:
# This function defines an objective for optimization, aiming to maximize R-
# Parameters:
#   - params: List of parameters to be optimized (in this case, only 'price'
#   - X_train: DataFrame of training features
#   - X_test: DataFrame of testing features
#   - y_train: Series of training target variable
#   - y_test: Series of testing target variable
# Returns:
#   - Negative R-squared as the objective value (to maximize R-squared)

def objective_function(params, X_train, X_test, y_train, y_test):
    # Extracting the 'price' parameter from the input
    price = params[0]

    # Setting the 'price' parameter for training features
    X_train['price'] = price

    # Training a Random Forest model on the modified training data
    model = train_random_forest(X_train, y_train, n_estimators=10, random_st

    # Making predictions on the testing data
    y_pred = predict_random_forest(X_test, model)

    # Calculating R-squared as the objective to be maximized
    residual_sum_of_squares = ((y_test - y_pred) ** 2).sum()
    total_sum_of_squares = ((y_test - y_test.mean()) ** 2).sum()
    r_squared = 1 - (residual_sum_of_squares / total_sum_of_squares)

    # Returning negative R-squared since we aim to maximize it in optimizati
    return -r_squared

# run_optimization:
# This function runs an optimization process to find the optimal 'price' par
# Parameters:
#   - X_train: DataFrame of training features
#   - X_test: DataFrame of testing features
#   - y_train: Series of training target variable
#   - y_test: Series of testing target variable
# Returns:
#   - optimal_price: The optimized 'price' parameter that maximizes R-square

def run_optimization(X_train, X_test, y_train, y_test):
    # Initial guess for the optimization
    initial_guess = [X_train['price'].mean()]

    # Running the optimization using Nelder-Mead method
    result = minimize(objective_function, initial_guess, args=(X_train.copy(

    # Extracting the optimized 'price' parameter from the result
    optimal_price = result.x[0]

    # Returning the optimal 'price' parameter
```

```
    # Returning the optimal 'price' parameter
    return optimal_price

def main():

    X, y = extract_features_and_target(df)
    X_train, X_test, y_train, y_test = training_testing_subsets(X, y, test_s
    optimal_price = run_optimization(X_train, X_test, y_train, y_test)

    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 67.66894407187863
```

In [ ]: