In [1]:
```python
import pandas as pd

# Replace the file path with your actual file path
file_path = "/Volumes/Untitled/aca2eb7d00ea1a7b8ebd4e68314663af.csv"

# Load the CSV file into a pandas DataFrame
df = pd.read_csv(file_path)

# Display the structure of the DataFrame
print("DataFrame Structure:")
print(df.info())

# Display the first few rows of the DataFrame
print("\nFirst few rows of the DataFrame:")
print(df.head())
```

```
DataFrame Structure:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 536 entries, 0 to 535
Data columns (total 4 columns):
 #   Column                         Non-Null Count  Dtype  
---  ------                         --------------  -----  
 0   order_item_id                  536 non-null    int64  
 1   shipping_limit_date            536 non-null    object 
 2   price                          536 non-null    float64
 3   product_category_name_english  536 non-null    int64  
dtypes: float64(1), int64(2), object(1)
memory usage: 16.9+ KB
None

First few rows of the DataFrame:
   order_item_id shipping_limit_date  price  product_category_name_english
0              1          2018-05-14   69.9                             39
1              1          2018-01-09   75.0                             39
2              1          2018-03-15   69.9                             39
3              1          2017-08-11   75.0                             39
4              2          2017-08-11   75.0                             39
```

In [5]:
```
# Pricing Optimization Problem with Gradient Boosting Algorithm

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

# Gradient Boosting is a machine learning algorithm used for solving
# pricing optimization problems. In this context, the algorithm sequentially
# builds decision trees to model the relationship between product prices and
# customer behavior. The objective function is designed to maximize the R-sq
# value, ensuring accurate predictions. The process involves iteratively fit
# decision trees, each correcting the errors of the previous ones. The 'opti
# function utilizes the Nelder-Mead optimization method to find the optimal
# parameter that maximizes the R-squared value, leading to an effective pric
# strategy. This approach allows businesses to adaptively adjust prices to
# enhance overall revenue and customer satisfaction.


import pandas as pd
import numpy as np
from scipy.optimize import minimize

# find_best_split:
# This function finds the best split point for a dataset based on variance r
# It takes two parameters: X - a DataFrame containing features, and y - a Se
# containing target values. It returns a tuple (best_split_feature, best_spl

def find_best_split(X, y):
    # Initialize variables to store the best split information
    best_split = None
    best_variance_reduction = 0

    # Iterate through each feature in the dataset
    for feature in X.columns:
        # Extract unique values of the current feature
        values = X[feature].unique()

        # Iterate through each unique value of the feature
        for value in values:
            # Create boolean masks for left and right subsets based on the c
            left_mask = X[feature] <= value
            right_mask = ~left_mask

            # Calculate variance for the left and right subsets
            left_variance = calculate_variance(y[left_mask])
            right_variance = calculate_variance(y[right_mask])

            # Calculate total variance for the entire dataset
            total_variance = len(y) * calculate_variance(y)

            # Calculate variance reduction for the current split
            variance_reduction = total_variance - (len(y[left_mask]) * left_

            # Update the best split if the current split provides greater va
            if variance_reduction > best_variance_reduction:
                best_variance_reduction = variance_reduction
                best_split = (feature, value)

    # Return the best split as a tuple (best_split_feature, best_split_value
    return best_split
```

```
# build_tree:
# This function recursively builds a decision tree based on the input datase
# It takes four parameters: X - a DataFrame containing features,
# y - a Series containing target values, max_depth - the maximum depth of th
# and depth - the current depth of the tree (used internally, default is 0).
# It returns a dictionary representing the decision tree.

def build_tree(X, y, max_depth, depth=0):
    # Base cases: if the maximum depth is reached or there is only one sampl
    if depth == max_depth or len(X) <= 1:
        return np.mean(y)

    # Find the best split for the current node
    split = find_best_split(X, y)

    # If no meaningful split is found, return the mean of the target values
    if split is None:
        return np.mean(y)

    # Extract feature and value from the split
    feature, value = split

    # Create boolean masks for left and right subsets based on the split
    left_mask = X[feature] <= value
    right_mask = ~left_mask

    # Recursively build the left and right subtrees
    left_subtree = build_tree(X[left_mask], y[left_mask], max_depth, depth +
    right_subtree = build_tree(X[right_mask], y[right_mask], max_depth, dept

    # Return a dictionary representing the current node and its subtrees
    return {'feature': feature, 'value': value, 'left': left_subtree, 'right

# predict_instance:
# This function predicts the target value for a single instance using a deci
# It takes two parameters: instance - a Pandas Series representing a single
# and tree - a dictionary representing the decision tree. It returns the pre

def predict_instance(instance, tree):
    # Base case: if the current node is a leaf node, return its value
    if isinstance(tree, (float, int)):
        return tree

    # Recursively traverse the tree based on the instance's feature values
    if instance[tree['feature']] <= tree['value']:
        return predict_instance(instance, tree['left'])
    else:
        return predict_instance(instance, tree['right'])

# predict_tree:
# This function predicts the target values for a set of instances using a de
# sion tree. It takes two parameters: X - a DataFrame containing features,
# and tree - a dictionary representing the decision tree. It returns an arra
# predicted target values.

def predict_tree(X, tree):
    # Initialize an empty list to store predictions
    predictions = []
```

```
    # Iterate through each instance in the dataset
    for _, instance in X.iterrows():
        # Predict the target value for the current instance using the given
        predictions.append(predict_instance(instance, tree))

    # Convert the list of predictions to a NumPy array and return it
    return np.array(predictions)

# extract_features_target:
# This function extracts learning variables and target-variable from a DataF
# It takes one parameter: df - a DataFrame containing relevant columns.
# It returns two objects: X - a DataFrame with training variables,
# and y - a Series representing the target variable.

def extract_features_target(df):
    # Extract relevant features and target from the DataFrame
    X = df[['order_item_id', 'price', 'product_category_name_english']]
    y = pd.to_datetime(df['shipping_limit_date']).dt.month

    # Return the extracted features and target
    return X, y

# split_data:
# This function splits the input features and target into training and testi
# It takes four parameters: X - a DataFrame containing features,
# y - a Series containing target values, test_size - the proportion of the d
# and random_state - the seed used by the random number generator for reproc
# It returns four objects: X_train, X_test - DataFrames with training and te
# and y_train, y_test - Series representing the target variables for trainin

def split_data(X, y, test_size=0.2, random_state=42):
    # Create a boolean mask for the test set based on the specified test_siz
    msk = np.random.rand(len(X)) < (1 - test_size)

    # Split the data into training and testing sets using the boolean mask
    X_train, X_test = X[msk], X[~msk]
    y_train, y_test = y[msk], y[~msk]

    # Return the split datasets
    return X_train, X_test, y_train, y_test

# objective_function:
# This function defines an objective function to be minimized for hyperparam
# It takes six parameters: params - a list of hyperparameters to be optimize
# X_train, X_test - DataFrames containing training and testing features,
# y_train, y_test - Series representing the target variables for training an
# and max_depth - the maximum depth for the decision tree.

def objective_function(params, X_train, X_test, y_train, y_test, max_depth):
    # Extract the price from the input parameters
    price = params[0]

    # Update the 'price' column in the training set with the extracted value
    X_train['price'] = price

    # Fit a decision tree using the modified training set
    tree = fit_decision_tree(X_train, y_train, max_depth)

    # Make predictions on the testing set using the trained decision tree
```

```
    # Make predictions on the testing set using the trained decision tree
    y_pred = predict_tree(X_test, tree)

    # Calculate the R-squared value for the predictions
    residuals = y_test - y_pred
    r_squared = 1 - np.sum(residuals ** 2) / np.sum((y_test - np.mean(y_test

    # Return the negation of R-squared as this is a minimization problem
    return -r_squared

# optimize_price:
# This function optimizes the price parameter using the Nelder-Mead method.
# It takes six parameters: initial_guess - the initial value for the optimiz
# X_train, X_test - DataFrames containing training and testing features,
# y_train, y_test - Series representing the target variables for training ar
# and max_depth - the maximum depth for the decision tree.
# It returns the optimized price parameter.

def optimize_price(initial_guess, X_train, X_test, y_train, y_test, max_dept
    # Use the Nelder-Mead method to minimize the objective function
    result = minimize(objective_function, initial_guess, args=(X_train.copy(

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

    # Return the optimized price parameter
    return optimal_price

# Auxiliary functions:

def fit_decision_tree(X, y, max_depth):
    return build_tree(X, y, max_depth)

def calculate_variance(y):
    return np.var(y)

def main():

    # Extract features and target variable
    X, y = extract_features_target(df)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = split_data(X, y)

    # Define the initial guess for the optimal price
    initial_guess = [df['price'].mean()]

    # Set the maximum depth for the decision tree
    max_depth = 3

    # Run the optimization
    optimal_price = optimize_price(initial_guess, X_train, X_test, y_train,

    # Display the optimal price
    print("Optimal Price:", optimal_price)

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

```
Optimal Price: 71.35858208955223
```

In [ ]: