```
import os
import pandas as pd
import numpy as np
from scipy.optimize import curve_fit, minimize, least_squares
import matplotlib.pyplot as plt

def profit_function(price, unit_cost, initial_demand, initial_price, elasticity):
    price = np.atleast_1d(price)
    price = np.clip(price, 0.5 * unit_cost, None)  # Assuming unit cost is a
reasonable minimum
    demand = initial_demand * ((price / initial_price) ** elasticity)
    profit = (price - unit_cost) * demand
    return profit.ravel()  # Ensure the return is 1-D array


def calculate_elasticity_coefficient(product_data):
    product_data.loc[:, 'Unit Cost'] = product_data['Unit Cost'].replace('[\$,]',
'', regex=True).astype(float)
    product_data.loc[:, 'Unit Price'] = product_data['Unit
Price'].replace('[\$,]', '', regex=True).astype(float)
    price_demand_data = product_data.groupby('Unit Price')['Order
Quantity'].sum().reset_index()

    def demand_model(x, a, b):
        return a * x**b

    popt, _ = curve_fit(demand_model, price_demand_data['Unit Price'],
price_demand_data['Order Quantity'])
    elasticity_coefficient = popt[1] * 25
    return elasticity_coefficient

def optimize_price_demand(product_data, elasticity_coefficient):
    initial_price = product_data['Unit Price'].iloc[0]
    initial_demand = product_data['Order Quantity'].sum()
    unit_cost = product_data['Unit Cost'].iloc[0]

    learning_rate = 0.01
    num_iterations = 1000
    prices, profits = [initial_price], []
    best_price, best_profit = initial_price, 0

    for _ in range(num_iterations):
        current_price = prices[-1]
        demand = initial_demand * ((current_price / initial_price) **
elasticity_coefficient)
        profit_gradient = demand + elasticity_coefficient * (current_price -
unit_cost) * demand / current_price
        updated_price = current_price + learning_rate * profit_gradient
        updated_profit = profit_function(updated_price, unit_cost, initial_demand,
initial_price, elasticity_coefficient)

        prices.append(updated_price)
```

```
        profits.append(updated_profit)

        if updated_profit > best_profit:
            best_profit = updated_profit
            best_price = updated_price

    return initial_price, best_price, best_profit, prices, profits

# Using price-based gradient descent optimization
def optimize_price_only(product_data, elasticity_coefficient):
    initial_price = product_data['Unit Price'].iloc[0]
    initial_demand = product_data['Order Quantity'].sum()
    unit_cost = product_data['Unit Cost'].iloc[0]

    learning_rate = 0.01
    num_iterations = 1000
    prices, profits = [initial_price], []
    best_price, best_profit = initial_price, 0

    for _ in range(num_iterations):
        current_price = prices[-1]
        profit_gradient = np.sum(-elasticity_coefficient * initial_demand *
((current_price / initial_price) ** (elasticity_coefficient - 1)))
        updated_price = current_price + learning_rate * profit_gradient
        updated_profit = profit_function(updated_price, unit_cost, initial_demand,
initial_price, elasticity_coefficient)

        prices.append(updated_price)
        profits.append(updated_profit)

        if updated_profit > best_profit:
            best_profit = updated_profit
            best_price = updated_price

    return initial_price, best_price, best_profit, prices, profits

def optimize_with_bfgs(product_data, elasticity_coefficient):
    initial_price = product_data['Unit Price'].iloc[0]
    prices, profits = [], []

    def callback(price):
        profit = profit_function(price, product_data['Unit Cost'].iloc[0],
product_data['Order Quantity'].sum(),
                                 product_data['Unit Price'].iloc[0],
elasticity_coefficient)
        prices.append(price[0])
        profits.append(profit[0])

    # Define the objective function to minimize
    def objective(price):
        return -profit_function(price, product_data['Unit Cost'].iloc[0],
product_data['Order Quantity'].sum(),
                                product_data['Unit Price'].iloc[0],
elasticity_coefficient)
```

```
    # Bounds ensure price does not go below half of unit cost or unreasonably high
    bounds = [(0.5 * product_data['Unit Cost'].iloc[0], None)]

    # Minimize using BFGS with bounds
    result = minimize(objective, [initial_price], method='L-BFGS-B',
bounds=bounds, callback=callback)
    best_price = result.x[0]
    best_profit = -result.fun

    return initial_price, best_price, best_profit, prices, profits

def optimize_with_kkt(product_data, elasticity_coefficient):
    initial_price = product_data['Unit Price'].iloc[0]
    prices, profits = [], []

    def callback(price):
        profit = profit_function(price, product_data['Unit Cost'].iloc[0],
product_data['Order Quantity'].sum(),
                                product_data['Unit Price'].iloc[0],
elasticity_coefficient)
        prices.append(price[0])
        profits.append(profit[0])

    # Define the objective function
    def objective(price):
        return -profit_function(price, product_data['Unit Cost'].iloc[0],
product_data['Order Quantity'].sum(),
                                product_data['Unit Price'].iloc[0],
elasticity_coefficient)

    # Define constraints
    cons = ({'type': 'ineq', 'fun': lambda price: price - 0.5 * product_data['Unit
Cost'].iloc[0]})

    # Minimize with constraints (KKT)
    result = minimize(objective, [initial_price], method='SLSQP',
constraints=cons, callback=callback)
    best_price = result.x[0]
    best_profit = -result.fun

    return initial_price, best_price, best_profit, prices, profits

# Update the analyze_product and analyze_all_products methods to include these
optimizations
def analyze_product(product_id, data_path, output_folder, plot=False,
display_plot=False, method_type='GM0'):
    sales_data = pd.read_csv(data_path)
    product_data = sales_data[sales_data['_ProductID'] == product_id]
    elasticity_coefficient = calculate_elasticity_coefficient(product_data)

    if method_type == 'BFGS':
        initial_price, best_price, best_profit, prices, profits =
optimize_with_bfgs(product_data, elasticity_coefficient)
```

```
        method_name = "BFGS"
    elif method_type == 'KKT':
        initial_price, best_price, best_profit, prices, profits =
optimize_with_kkt(product_data, elasticity_coefficient)
        method_name = "KKT"
    elif method_type == "GM0":
        initial_price, best_price, best_profit, prices, profits =
optimize_price_only(product_data, elasticity_coefficient)
        method_name = "GradiantMethods_PriceOnly"
    elif method_type == "GM1":
        initial_price, best_price, best_profit, prices, profits =
optimize_price_demand(product_data, elasticity_coefficient)
        method_name = "GradiantMethods_PriceAndDemand"
    else:
        return "Wrong typed: method type"

    # Plotting logic if needed
    if plot:
        plt.figure(figsize=(12, 6))
        plt.subplot(1, 2, 1)
        plt.plot(prices, marker='o', linestyle='-')
        plt.title(f'Price Optimization Over Iterations ({method_name})')
        plt.xlabel('Iteration')
        plt.ylabel('Price ($)')
        plt.grid(True)

        plt.subplot(1, 2, 2)
        plt.plot(profits, marker='o', linestyle='-', color='red')
        plt.title(f'Profit Optimization Over Iterations ({method_name})')
        plt.xlabel('Iteration')
        plt.ylabel('Profit ($)')
        plt.grid(True)
        plt.tight_layout()
        plt.draw()


plt.savefig(f'{output_folder}/Product_{product_id}_{method_name}_Iteration.png')
        if display_plot:
            plt.show()
        plt.close()


    return product_data['Unit Price'].iloc[0], best_price, best_profit,
method_name

def analyze_all_products(data_path, output_folder, method_type="GM0",
display_plots=False):
    # Create method-specific output folder
    method_output_folder = os.path.join(output_folder, method_type)
    if not os.path.exists(method_output_folder):
        os.makedirs(method_output_folder)

    sales_data = pd.read_csv(data_path)
    product_ids = sales_data['_ProductID'].unique()
```

```
    results = []

    for product_id in product_ids:
        initial_price, best_price, best_profit, method_name = analyze_product(
            product_id, data_path, method_output_folder, plot=True,
display_plot=display_plots, method_type=method_type)
        results.append([product_id, initial_price, best_price, best_profit,
method_name])

    results_df = pd.DataFrame(results, columns=['ProductID', 'Initial Price',
'Best Price', 'Best Profit', 'Method'])

results_df.to_csv(f'{method_output_folder}/All_Products_Optimization_{method_name}
.csv')

def describe_methods():
    print("Available Optimization Methods and their Usage:")
    print("------------------------------------------------")
    print("1. Gradient Method (GM):")
    print("   - 'GM0' for PriceOnly Gradient Descent Optimization")
    print("   - 'GM1' for Demand and Price Gradient Descent Optimization")
    print("   Use these methods to optimize price based on gradient descent
technique.")

    print("\n2. BFGS:")
    print("   - 'BFGS' for Broyden-Fletcher-Goldfarb-Shanno optimization
algorithm")
    print("   Use BFGS to find the local maximum of profit function using a quasi-
Newton method.")

    print("\n3. Karush-Kuhn-Tucker (KKT):")
    print("   - 'KKT' for optimization using Karush-Kuhn-Tucker conditions")
    print("   Use KKT when there are constraints that the solution needs to
satisfy.")

    print("\nFunction Usage:")
    print("---------------")
    print("analyze_product(product_id, data_path, output_folder, plot=False,
display_plot=False, method_type='GM0')")
    print(" - product_id: ID of the product to analyze.")
    print(" - data_path: Path to the CSV file containing product data.")
    print(" - output_folder: Directory to save the output plots and data files.")
    print(" - plot: Set to True to generate plots.")
    print(" - display_plot: Set to True to display plots during execution.")
    print(" - method_type: Specifies the optimization method to use.")
    print("   Options are 'GM0', 'GM1', 'BFGS', 'KKT'.")

    print("\nanalyze_all_products(data_path, output_folder, method_type='GM0',
display_plots=False)")
    print(" - data_path: Path to the CSV file containing all products data.")
    print(" - output_folder: Directory to save the output plots and data files for
all products.")
    print(" - method_type: Specifies the optimization method to use for all
products.")
```

```
    print(" - display_plots: Set to True to display plots during execution.")
    print("   Options are 'GM0', 'GM1', 'BFGS', 'KKT'.")
    print("\nUse these functions to analyze product profitability under various
pricing scenarios using different optimization methods.")

# usage introduction:
describe_methods()
```

```

Available Optimization Methods and their Usage:
------------------------------------------------
1. Gradient Method (GM):
   - 'GM0' for PriceOnly Gradient Descent Optimization
   - 'GM1' for Demand and Price Gradient Descent Optimization
   Use these methods to optimize price based on gradient descent technique.

2. BFGS:
   - 'BFGS' for Broyden-Fletcher-Goldfarb-Shanno optimization algorithm
   Use BFGS to find the local maximum of profit function using a quasi-Newton
method.

3. Karush-Kuhn-Tucker (KKT):
   - 'KKT' for optimization using Karush-Kuhn-Tucker conditions
   Use KKT when there are constraints that the solution needs to satisfy.

Function Usage:
---------------
analyze_product(product_id, data_path, output_folder, plot=False,
display_plot=False, method_type='GM0')
 - product_id: ID of the product to analyze.
 - data_path: Path to the CSV file containing product data.
 - output_folder: Directory to save the output plots and data files.
 - plot: Set to True to generate plots.
 - display_plot: Set to True to display plots during execution.
 - method_type: Specifies the optimization method to use.
   Options are 'GM0', 'GM1', 'BFGS', 'KKT'.

analyze_all_products(data_path, output_folder, method_type='GM0',
display_plots=False)
 - data_path: Path to the CSV file containing all products data.
 - output_folder: Directory to save the output plots and data files for all
products.
 - method_type: Specifies the optimization method to use for all products.
 - display_plots: Set to True to display plots during execution.
   Options are 'GM0', 'GM1', 'BFGS', 'KKT'.

Use these functions to analyze product profitability under various pricing
scenarios using different optimization methods.
```

```
analyze_all_products('US_Regional_Sales_Data.csv', 'result_images', "GM0")
```

```
analyze_all_products('US_Regional_Sales_Data.csv', 'result_images', "GM1")
```

```
analyze_all_products('US_Regional_Sales_Data.csv', 'result_images', "BFGS")
```

```
analyze_all_products('US_Regional_Sales_Data.csv', 'result_images', "KKT")
```

```

```