Homework 2

Please submit the solution to gradescope by 11:59 PM, Sept 26, Thursday.

Name: Ivy Nangalia

PID: 730670491

Problem 1 Sales Data Analysis (35 Points)

Dataset: You will work with a simulated dataset representing the monthly sales data for a fictional company over a period of 3 years (36 months). The dataset is provided as a NumPy array.

1.0 . Setup and Initialization

Import NumPy and initialize the dataset as a NumPy array.

In [ ]:
```
import numpy as np

sales_data = np.array([207, 217, 244, 269, 247, 248, 273, 260, 243, 265, 26
                       310, 296, 313, 340, 336, 356, 335, 327, 374, 358, 37
                       405, 432, 425, 455, 443, 446, 439, 477, 454, 449, 49
```

1.1 Monthly Analysis (15 Points)

Using the `sales_data` array, perform the following tasks:

1. Calculate the total sales for each year, and store the result in a NumPy array `total_sales_per_year`. Display the `total_sales_per_year`. (2 points)

2. Calculate the average monthly sales for each year, and store the result in a NumPy array `average_monthly_sales_per_year`. Display the `average_monthly_sales_per_year`. (2 points)

3. Calculate the average season sales for each year, and store the result in a `3` by `4` NumPy array `average_season_sales_per_year`. Each row represent a year, and each column represent a season. Display the `average_season_sales_per_year`. (6 points)

4. Identify the month with the highest sales and the month with the lowest sales on average over the three year period. Store the numerical representation (1-12) of the month in the variables `max_sales_month` and `min_sales_month`. (5 points)

You can assume the years are in chronological order. For example, the first value represents sales in January of Year 1 and the last value represents sales in December of Year 3.

In [ ]:
```
sales_data = sales_data.reshape(3,12)
total_sales_per_year = sales_data.sum(axis=1)
total_sales_per_year
```

Out[ ]:
```
array([3016, 4093, 5401])
```

In [ ]:
```
average_monthly_sales_per_year = np.mean(sales_data, axis=1)
average_monthly_sales_per_year
```

Out[ ]:
```
array([251.33333333, 341.08333333, 450.08333333])
```

In [ ]:
```
average_season_sales_per_year = np.mean(sales_data.reshape(3, 4, 3), axis=2
average_season_sales_per_year
```

Out[ ]:
```
array([[222.66666667, 254.66666667, 258.66666667, 269.33333333],
       [306.33333333, 344.        , 345.33333333, 368.66666667],
       [420.66666667, 448.        , 456.66666667, 475.        ]])
```

In [ ]:
```
average_per_month = np.mean(sales_data, axis=0)
highest_month = average_per_month.argmax() + 1 # to convert to the month nu
lowest_month = average_per_month.argmin() + 1
# printing the name of the month for fun
months: dict = {
    1: "January", 2: "February", 3: "March", 4: "April", 5: "May", 6: "June
    7: "July", 8: "August", 9: "September", 10: "October", 11: "November",
                }

print(f"{months[highest_month]} had the highest average monthly sales (${st
      f"while {months[lowest_month]} had the lowest average sales monthly (
```

```
December had the highest average monthly sales ($378.0), while January had t
he lowest average sales monthly ($307.3333333333333).
```

1.2 Growth Rate Calculation (5 Points)

Write a function `calculate_growth_rate(data)` that takes the `sales_data` array as input and returns the growth rates for each month in a NumPy array. You can assume the growth rate for the first month is 0.01.

Calculate the monthly growth rate of sales using the formula:

$$\text{growthrate}[i] = \frac{\text{salesdata}[i] - \text{salesdata}[i-1]}{\text{salesdata}[i-1]}$$

Display the result of `calculate_growth_rate(sales_data)`.

In [ ]:
```
def calculate_growth_rate(data):
    shape = data.shape
    data = data.flatten()
    rate_array = np.zeros(len(data))
    rate_array[0] = 0.01
    for i in range(1, len(data)):
        rate_array[i] = (data[i] - data[i-1])/data[i-1]
    return rate_array.reshape(shape)
```

In [ ]:
```
rates = calculate_growth_rate(sales_data)
rates
```

Out[ ]:
```
array([[ 0.01      ,  0.04830918,  0.12442396,  0.10245902, -0.08178439,
         0.00404858,  0.10080645, -0.04761905, -0.06538462,  0.09053498,
        -0.00754717,  0.06463878],
       [ 0.10714286, -0.04516129,  0.05743243,  0.08626198, -0.01176471,
         0.05952381, -0.05898876, -0.0238806 ,  0.14373089, -0.04278075,
         0.05027933, -0.0106383 ],
       [ 0.08870968,  0.06666667, -0.0162037 ,  0.07058824, -0.02637363,
         0.00677201, -0.01569507,  0.08656036, -0.04821803, -0.01101322,
         0.10022272, -0.0242915 ]])
```

1.3 Growth Rate Summary (5 Points)

Identify the following months:

- The months in the past three years saw the largest increase and the largest decrease in sales (i.e. from January to February of Year 1 we saw the largest decrease in sales)
- The month on average with the largest increase and largest decrease in sales

Show both the python code and conclusion.

In [ ]:
```
rates = rates.reshape(-1, 12) # reshaping to use indecies for months and ye
year_month_array = np.array([range(1,37)])
year_month_array = year_month_array.reshape(3,12)
largest_increase = rates.argmax() + 1
largest_decrease = rates.argmin() + 1

def month_scale(month: int):
    year: int = 1
    while month > 12:
        month -= 12
        year += 1
```

```
    return (month, year)

largest_increase_month = month_scale(largest_increase)
largest_decrease_month = month_scale(largest_decrease)

print(f"We saw the largest sales increase from {months[largest_increase_mon
      f" of Year {largest_increase_month[1]}.")

print(f"We saw the largest sales decrease from {months[largest_decrease_mon
      f" of Year {largest_decrease_month[1]}.")
```

```
We saw the largest sales increase from August to September of Year 2.
We saw the largest sales decrease from April to May of Year 1.
```

In [ ]:

```
# average monthly values
average_rates_monthly = np.mean(rates, axis=0)
highest_average_month = average_rates_monthly.argmax() + 1
lowest_average_month = average_rates_monthly.argmin() + 1
print("Across all years, on average, " +
    f"{months[highest_average_month]} saw the largest average increase in s
      f"while {months[lowest_average_month]} saw the largest average decrea
```

```
Across all years, on average, April saw the largest average increase in sale
s ($+0.08643641083941041), while May saw the largest average decrease in sal
es ($-0.0399742396243599).
```

1.4 Moving Average (5 points)

Calculate the 3-month moving average of the `sales_data` and store the result in a NumPy array called `moving_average`. The calculation starts from the third month in Year 1.

The three-period moving average for month t is calculated as:

$$\mathrm{MA}_3(t) = \frac{x_t + x_{t-1} + x_{t-2}}{3}.$$

Where:

- x_t is the sales in month t,
- x_{t-1} is the sales in month $t-1$,
- x_{t-2} is the sales in month $t-2$.

Display the `moving_average`.

In [ ]:

```
def calc_moving_average(data):
    shape = data.shape
    data = data.flatten()
    moving_averages = np.zeros(len(data))
    for i in range(2, len(data)):
```

```
        moving_averages[i] = (data[i] + data[i - 1] + data[i - 2]) / 3
    return moving_averages.reshape(shape)

moving_average = calc_moving_average(sales_data)
moving_average
```

Out[ ]:

```
array([[  0.        ,   0.        , 222.66666667, 243.33333333,
        253.33333333, 254.66666667, 256.        , 260.33333333,
        258.66666667, 256.        , 257.        , 269.33333333],
       [284.33333333, 295.33333333, 306.33333333, 316.33333333,
        329.66666667, 344.        , 342.33333333, 339.33333333,
        345.33333333, 353.        , 369.33333333, 368.66666667],
       [384.33333333, 403.        , 420.66666667, 437.33333333,
        441.        , 448.        , 442.66666667, 454.        ,
        456.66666667, 460.        , 465.66666667, 475.        ]])
```

1.5 Sales Anomaly Detection (5 points)

Write a function `sales_anomaly_detection(sales_data)` that takes the `sales_data` array as input, identify months where sales were significantly higher or lower than the previous month's sales (more than 12% change), and returns a dictionary mapping the month number (1-36) to the sales data for all anomalous months.

Display the result of `sales_anomaly_detection(sales_data)`.

In [ ]:

```
def sales_anomaly_detection(data):
    anomalies = {}
    rates = calculate_growth_rate(data)
    for i in range(0, len(rates)):
        if abs(rates[i]) > 0.12:
            anomalies[i] = rates[i]
    return anomalies
```

In [ ]:

```
def sales_anomaly_detection(data):
    anomalies = {}
    rates = calculate_growth_rate(data).flatten()
    rate_indecies = np.argsort(abs(rates))[::-1]
    for index in rate_indecies:
        if rates[index] > 0.12:
            anomalies[index] = rates[index]
    return anomalies
```

In [ ]:

```
print(sales_anomaly_detection(sales_data))
```

```
{20: 0.1437308868501529, 2: 0.12442396313364056}
```

Problem 2: Analysis on the diamonds dataset (65 points)

Here is a brief description of the key columns in the `diamonds` dataset from Seaborn:

`Carat` : The weight of the diamond (continuous variable). Larger diamonds have higher carat values.

`Cut` : The quality of the diamond's cut (categorical variable).

`Color` : The diamond's color grade (categorical variable).

`Clarity` : The clarity of the diamond (categorical variable), which indicates how free the diamond is from internal flaws (inclusions) or external blemishes. It ranges from `IF` (Internally Flawless) to `I3` (Included).

`Depth` : The total depth percentage (continuous variable). It's the ratio of the depth of the diamond to its average diameter.

`Table` : The width of the diamond's top relative to its widest point (expressed as a percentage).

`Price` : The price of the diamond in US dollars (continuous variable).

`x` : Length of the diamond in millimeters (continuous variable).

`y` : Width of the diamond in millimeters (continuous variable).

`z` : Depth of the diamond in millimeters (continuous variable).

This dataset provides valuable features for exploring the relationship between various attributes and the price of diamonds.

2.1 Load the dataset from a `diamonds.csv`. (1 point)

2.2 How many different levels are there in the `cut` column? Provide the names of all levels. (4 points)

2.3 Create three new columns, `x_in_inch`, `y_in_inch`, and `z_in_inch`, which convert the units of the original `x`, `y`, and `z` from millimeters to inches. Print the new table. (5 points)

2.4 Create a new column called `Normalized_depth`, which scales the depth values between 0 and 1. (10 points)

The equation to normalize the `depth` column is given as:

$$\text{depth}_{\text{normalized}} = \frac{\text{depth} - \min(\text{depth})}{\max(\text{depth}) - \min(\text{depth})}$$

Where:

- depth is the original value,
- $\min(\text{depth})$ is the minimum value in the `depth` column,
- $\max(\text{depth})$ is the maximum value in the `depth` column.

Print the new table.

2.5 Select the rows from the entire table where `clarity` is `SI2`. Drop the columns `color` and `clarity`. Print the selected table. (5 points)

2.6 Suppose we have a linear model that predicts the price of a diamond using the `carat` value:

$$\text{Predicted Price} = 7769 \times \text{carat} - 2262.$$

In the selected table from 2.5, create a new column called `Predicted_Price`, which contains the predicted price using the above formula. Display the new table. (10 points)

2.7 In the new table from 2.6, calculate the difference between the actual price and the predicted price. How many rows have a prediction error that is smaller than 20% of the actual price? (5 points)

2.8 Sort the table in 2.7 by 'carat' column in increasing order. Display the first 3 rows and last 3 rows of the sorted table. (5 points). Concatenate the first 3 rows and last 3 rows into a new table. (5 points) (10 points in total)

2.9 In the table from 2.7, what is the value of `carat` that has the smallest prediction error? What is the value of `carat` that has the largest prediction error. (5 points)

2.10 In the orignal entire table, calculate the average prices for each level of cut. Display the result as a pd.Series. Does a better cut lead to a higher price? (10 points)

In [ ]:

```
import pandas as pd
diamonds = pd.read_csv("diamonds.csv")
diamonds.head()
```

Out[ ]:

	Carat	Cut	Color	Clarity	depth	table	Price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

In [ ]:

```
diamonds["Cut"].unique()
```

Out[ ]:

```
array(['Ideal', 'Premium', 'Good', 'Very Good', 'Fair'], dtype=object)
```

In [ ]:

```
diamonds["x_in_inch"] = diamonds["x"].transform(lambda x: x/25.5)
diamonds["y_in_inch"] = diamonds["y"].transform(lambda x: x/25.5)
diamonds["z_in_inch"] = diamonds["z"].transform(lambda x: x/25.5)
diamonds
```

Out[ ]:

	Carat	Cut	Color	Clarity	depth	table	Price	x	y	z	x_in_in
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	0.1549
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	0.1525
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	0.1588
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	0.1647
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	0.1701
...	...	...	...	...	...	...	...	...	...	...	
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50	0.2254
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61	0.2231
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56	0.2219
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74	0.2411
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64	0.2286

53940 rows × 13 columns

In [ ]:

```
diamonds["Normalized_depth"] = ((
    diamonds["depth"] - diamonds["depth"].min())
    /
    (diamonds["depth"].max() - diamonds["depth"].min())
    )
```

```
diamonds
```

Out[ ]:

	Carat	Cut	Color	Clarity	depth	table	Price	x	y	z	x_in_in
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	0.1549
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	0.1525
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	0.1588
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	0.1647
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	0.1701
...	...	...	...	...	...	...	...	...	...	...	
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50	0.2254
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61	0.2231
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56	0.2219
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74	0.2411
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64	0.2286

53940 rows × 14 columns

In [ ]:

```
# selecting SI2 clarity rows
SI2_df = diamonds[diamonds['Clarity'] == 'SI2']
SI2_df = SI2_df.drop(labels=["Color", "Clarity"], axis=1)
```

In [ ]:

```
# creating the prediction price
SI2_df["Predicted_Price"] = (7769 * SI2_df["Carat"]) - 2262
SI2_df
```

Out[ ]:

	Carat	Cut	depth	table	Price	x	y	z	x_in_inch
0	0.23	Ideal	61.5	55.0	326.0	3.95	3.98	2.43	0.154902
4	0.31	Good	63.3	58.0	335.0	4.34	4.35	2.75	0.170196
13	0.31	Ideal	62.2	54.0	344.0	4.35	4.37	2.71	0.170588
14	0.20	Premium	60.2	62.0	345.0	3.79	3.75	2.27	0.148627
16	0.30	Ideal	62.0	54.0	348.0	4.31	4.34	2.68	0.169020
...	...	...	...	...	...	...	...	...	...
53915	0.77	Ideal	62.1	56.0	2753.0	5.84	5.86	3.63	0.229020
53928	0.79	Premium	61.4	58.0	2756.0	6.03	5.96	3.68	0.236471
53938	0.86	Premium	61.0	58.0	2757.0	6.15	6.12	3.74	0.241176
53939	0.75	Ideal	62.2	55.0	2757.0	5.83	5.87	3.64	0.228627
Predicted_Price	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

9195 rows × 13 columns

In [ ]:

```
# creating prediction error
SI2_df["Prediction_Error"] = abs(SI2_df["Price"] - SI2_df["Predicted_Price"

print(f"{len(SI2_df[SI2_df["Prediction_Error"] < 0.2])}"
      + " rows have a prediction error that is smaller than 20% of the actu
```

```
2169 rows have a prediction error that is smaller than 20% of the actual pri
ce.
```

In [ ]:

```
sorted_table = SI2_df["Carat"].sort_values(ascending=True)
result = pd.concat([sorted_table.head(3), sorted_table.dropna().tail(3)])
# dropna because 2/3 of the last values are NaN
result
```

Out[ ]:

```
14       0.20
43989    0.21
0        0.23
27518    3.01
26100    3.04
27638    3.04
Name: Carat, dtype: float64
```

In [ ]:

```
carats = SI2_df["Prediction_Error"].groupby(SI2_df["Carat"]).mean()
carats = carats.sort_values()
carats_concat = pd.concat([carats.head(3), carats.tail(3)])
carats_concat
```

Out[ ]:
```
Carat
2.67    0.010958
2.42    0.020053
2.57    0.023314
0.23    2.257825
0.21    2.600279
0.20    3.052754
Name: Prediction_Error, dtype: float64
```

In [ ]:
```python
cut = diamonds["Price"].groupby(diamonds["Cut"]).mean()
cut.sort_values(ascending=False)
```

Out[ ]:
```
Cut
Premium      4584.257704
Fair         4358.757764
Very Good    3981.759891
Good         3928.864452
Ideal        3457.541970
Name: Price, dtype: float64
```

A better cut does not necessarily lead to a higher price. Although the "Premium" diamonds have the highest average price, it's led by "Fair" diamonds, which is not the next best cut.