Statistical Foundations: Practical Assignment 1

Submission Info

Attribute	Value
Name	Ayushkar Pau
ID	GF202343142
Subject	Statistical Foundation of Data Science
Assignment	Practical 1 - Testing Pandas and Numpy
Repo	View my GitHub Repo

Assignment Overview

This notebook contains the solution for the first practical assignment in the Statistical Foundation of Data Sciences course. It covers key data analysis tasks such as calculating statistical measures, handling missing values, and performing array manipulations with NumPy.

Notebook Introduction

This notebook tackles the four core problems for the first practical assignment. We will use a synthetic dataset to perform a series of statistical analyses and array manipulations.

Key Tasks to be Performed:

- **Task 1: Central Tendency Calculation** We will compute the mean, median, and a custom age-weighted mean for the income data, ensuring we handle missing values correctly.
- **Task 2: Data Standardization and Outlier Detection** This task involves standardizing the income data into z-scores and identifying outliers based on the `|z| > 3` rule.

- **Task 3: Data Binning and Aggregation** We will segment the data into specific age bins and compute the count, mean income, and median score for each group.
- **Task 4: NumPy Array Operations** The final section will demonstrate fundamental NumPy operations, including reshaping, indexing, broadcasting, and linear algebra functions like calculating a determinant and inverse.

General Instructions & Setup

As per the assignment requirements, this notebook will adhere to the following:

1. A synthetic dataset with `NaN` values will be used.
2. The initial random seed is set to `42` for reproducibility.
3. All `NaN` values are handled appropriately without unnecessary data loss.

Let's begin with the Environment setup and move to the problem-1.

Environment Setup and Dependencies

Start by importing all the required libraries and generating the synthetic dataset for the assignment.

In [1]:
```
# Import all necessary libraries
import pandas as pd
import numpy as np

# Set the random seed to 42 for reproducibility
np.random.seed(42)

print("Libraries imported successfully!")
print(f"Pandas version: {pd.__version__}")
print(f"NumPy version: {np.__version__}")
```

```
Libraries imported successfully!
Pandas version: 2.2.3
NumPy version: 2.2.2
```

Create Synthetic Data

The following code cell generates the synthetic dataset that will be used for all the problems in this assignment.

In [2]:
```python
# Generate the synthetic dataset with 100 records
num_records = 100
data = {
    'age': np.random.randint(18, 65, size=num_records),
    'income': 50000 + np.random.normal(0, 15000, size=num_records),
    'score': np.random.uniform(0, 100, size=num_records)
}

# Create a pandas DataFrame
df = pd.DataFrame(data)

# Introduce some NaN (Not a Number) values into the dataset
nan_indices_income = np.random.choice(df.index, size=int(num_records * 0.1), r
df.loc[nan_indices_income, 'income'] = np.nan

nan_indices_score = np.random.choice(df.index, size=int(num_records * 0.1), re
df.loc[nan_indices_score, 'score'] = np.nan

# Display the first 5 rows and a summary to verify the data
print("--- First 5 Rows of the Dataset ---")
print(df.head())
print("\n--- Dataset Information (Checking for NaNs) ---")
df.info()
```

```
--- First 5 Rows of the Dataset ---
   age        income      score
0   56  59544.576625        NaN
1   46  36399.189971  98.216834
2   32  57140.638811  51.663589
3   60  69554.919026  26.082917
4   25  53173.805185  99.625370

--- Dataset Information (Checking for NaNs) ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   age     100 non-null    int64
 1   income  90 non-null     float64
 2   score   90 non-null     float64
dtypes: float64(2), int64(1)
memory usage: 2.5 KB
```

Problem 1: Measures of Central Tendency

> **Instruction**: Compute (a) mean, (b) median, and (c) age-weighted mean of income. Ignore NaNs where appropriate. Explain when a weighted mean is preferable.

1.1: Calculating Mean, Median and Age-weighted mean income:

First, we'll calculate the standard mean and median for the income column. The pandas functions mean() and median() will be used, as they automatically handle the NaN values present in the data. Them we will exclude the NaN values to calculate Age-weighted mean income.

In [3]:
```
# (a) Mean income - pandas handles NaNs automatically
mean_income = df['income'].mean()
print(f"(a) Mean income: ${mean_income:,.2f}")

# (b) Median income - pandas handles NaNs automatically
median_income = df['income'].median()
print(f"(b) Median income: ${median_income:,.2f}")

# (c) Age-weighted mean income
# First, create a temporary DataFrame that excludes rows with NaN income
df_no_nan_income = df.dropna(subset=['income'])

# Use the efficient numpy.average() function with the weights parameter
weighted_mean_income = np.average(
    df_no_nan_income['income'],
    weights=df_no_nan_income['age']
)
print(f"(c) Age-weighted mean income: ${weighted_mean_income:,.2f}")

print(f"\nCalculations are based on {df['income'].notna().sum()} valid income
```

```
(a) Mean income: $50,506.93
(b) Median income: $49,051.09
(c) Age-weighted mean income: $50,729.18

Calculations are based on 90 valid income records.
```

Solution Methodology

My approach to solving this problem was as follows:

1. **Mean and Median Calculation:** I utilized the standard pandas `.mean()` and `.median()` methods directly on the `income` series. These functions are efficient as they automatically ignore `NaN` values.
2. **Data Preparation for Weighted Mean:** To prepare for the weighted mean calculation, I created a temporary, clean DataFrame by dropping only the rows where `income` was missing. This ensured the age and income data aligned perfectly.
3. **Age-Weighted Mean Calculation:** I used the `numpy.average()` function, which is specifically designed for these calculations. I passed the income data as the main variable and the corresponding age data

to the `weights` parameter.

Analysis: When is a Weighted Mean Preferable?

A weighted mean is a better tool than a simple mean when some of your data points are more significant than others. A simple mean treats every value as having equal importance, which isn't always the case in the real world.

For example, in our dataset, we used age as a weight. This gives more influence to the income of older individuals. This might be useful if we believe that older individuals have more established careers and their income is a more stable indicator. Other common examples include calculating a student's final grade, where a final exam has more weight than unit-1, or analyzing an investment portfolio, where larger investments have a bigger impact on the overall return.

Problem 2: Standardization and Outlier Detection

> **Instruction**: Standardize income using z-score and identify outliers using the |z| > 3 rule. Handle NaNs correctly without dropping entire rows.

What is a Z-Score?

A **z-score** is a standardized value that tells us how many standard deviations a specific data point is from the mean of the dataset. The formula is: $$ z = \frac{(x - \mu)}{\sigma} $$ Where x is the data point, μ is the mean, and σ is the standard deviation.

This is extremely useful for finding outliers. A common rule is that any data point with an absolute z-score of more than 3 is considered an **outlier**, because it is very far from the average value. We will apply this rule to our income data.

In [4]:
```
print("=== PROBLEM 2: Standardization and Outlier Detection ===\n")

# 1. Calculate the mean and standard deviation, which are needed for the z-sco
#    Pandas automatically ignores NaNs for these calculations.
income_mean = df['income'].mean()
income_std = df['income'].std()

# 2. Standardize the income column and add it as a new 'income_zscore' column.
#    The formula is z = (value - mean) / std.
#    This operation correctly handles NaNs; a NaN income results in a NaN z-sc
```

```python
df['income_zscore'] = (df['income'] - income_mean) / income_std

# 3. Identify outliers using the |z| > 3 rule.
#    .abs() gets the absolute value, and we find rows where it's > 3.
outliers_df = df[df['income_zscore'].abs() > 3]
outlier_count = len(outliers_df)

# 4. Report the final results.
print(f"Number of income outliers found (|z| > 3): {outlier_count}")

# Display the outlier rows if any were found.
if outlier_count > 0:
    print("\n--- Detected Outlier Rows ---")
    # We display only the most relevant columns for a clean output.
    print(outliers_df[['age', 'income', 'income_zscore']])
else:
    print("\nNo outliers were detected based on the |z| > 3 rule.")

# Show the first few rows of the updated DataFrame to see the new column.
print("\n--- DataFrame Head with New Z-Score Column ---")
print(df.head())
```

```
=== PROBLEM 2: Standardization and Outlier Detection ===

Number of income outliers found (|z| > 3): 0

No outliers were detected based on the |z| > 3 rule.

--- DataFrame Head with New Z-Score Column ---
   age        income      score  income_zscore
0   56  59544.576625        NaN       0.585352
1   46  36399.189971  98.216834      -0.913733
2   32  57140.638811  51.663589       0.429654
3   60  69554.919026  26.082917       1.233704
4   25  53173.805185  99.625370       0.172729
```

Solution Methodology for Problem 2

My approach for standardizing the data and identifying outliers was as follows:

1. **Calculate Core Statistics**: I began by calculating the **mean** and **standard deviation** of the `income` column. I used the built-in pandas methods, which conveniently ignore any `NaN` values during this calculation.

2. **Compute Z-Scores**: I then created a new column in the DataFrame called `income_zscore`. This was done by applying the z-score formula in a single vectorized operation. This efficient method also ensures that any `NaN` in the original `income` column results in a `NaN` in the `income_zscore` column, fulfilling the requirement to handle missing

values properly.

3. **Filter for Outliers**: To find the outliers, I used **boolean masking**. I first took the absolute value of the `income_zscore` column and then filtered the DataFrame to keep only the rows where this value was greater than 3.

4. **Count and Report**: Finally, I counted the number of rows in the filtered outlier DataFrame to get the final count. The result was presented in a clean summary, and the actual outlier rows were displayed if any were found.

Problem 3: Age Binning and Aggregation

> **Instruction**: Create age bins `[18-25)`, `[25-35)`, `[35-45)`, `[45-60)` and compute for each bin: count of observations, mean income, and median score. Present results in a tidy DataFrame sorted by age bin.

What is Binning and Aggregation?

Binning is the process of converting continuous numerical data, like age, into discrete groups or "bins." Instead of analyzing every single age, we can look at patterns across broader age ranges (e.g., 18-24, 25-34). This makes it easier to spot trends that might be hidden in the raw data.

Once the data is binned, we can perform an **aggregation**. This simply means we calculate a summary statistic (like a `count`, `mean`, or `median`) for each of those bins. For this problem, we will group our data by the new age bins and then aggregate the required statistics to see how income and scores differ across age groups.

In [5]:
```
print("=== PROBLEM 3: Age Binning and Aggregation ===\n")

# 1. Define the age bins and labels.
bins = [18, 25, 35, 45, 60]
labels = ['18-24', '25-34', '35-44', '45-59']

# 2. Create the 'age_group' column.
df['age_group'] = pd.cut(df['age'], bins=bins, labels=labels, right=False)

# 3. Group by the new age bins and aggregate.
#    We add observed=False to silence the warning and maintain the current beh
binned_stats = df.groupby('age_group', observed=False).agg(
```

```
    count_of_observations=('age', 'count'),
    mean_income=('income', 'mean'),
    median_score=('score', 'median')
).round(2)

# 4. Display the final tidy DataFrame.
print("--- Tidy DataFrame of Binned Statistics ---")
print(binned_stats)
```

```
=== PROBLEM 3: Age Binning and Aggregation ===

--- Tidy DataFrame of Binned Statistics ---
           count_of_observations  mean_income  median_score
age_group
18-24                         15     45806.70         76.86
25-34                         21     47585.24         50.11
35-44                         26     59198.60         48.60
45-59                         24     48548.52         40.45
```

Solution Methodology for Problem 3

My strategy for this problem was to leverage the built-in data transformation and aggregation tools in pandas to produce the required tidy DataFrame.

1. **Bin and Label Definition**: I started by creating two Python lists: one to define the numerical edges of the age bins and another to hold the corresponding string labels for clear presentation. I set the `right=False` parameter in my approach to ensure the intervals were left-inclusive as required (e.g., `[18-25)`).

2. **Categorization with `pd.cut`**: I used the `pd.cut()` function to efficiently segment the continuous 'age' data into the discrete categories I had defined, creating a new `age_group` column in the process.

3. **One-Step Aggregation**: The core of my solution was a single `groupby('age_group').agg()` command. This powerful method allowed me to group the entire DataFrame by the new age bins and compute all three required statistics (count, mean income, and median score) simultaneously. This approach is not only concise but also correctly handles any missing values in the data. The resulting DataFrame was already sorted and formatted as required.

Problem 4: Array Operations and Linear Algebra

> **Instruction**: Create an array it cannot be of 1 Dimension. And then

> showcase the operation for the following:
>
> - Shape and Resize
> - shape, size, Transpose, Flatten
> - Showcasing negative indexing and display error while doing slicing
> - Arithmetic Operations → Broadcasting, Dot Product
> - Linear Algebra → Determinant, Inverse

Key NumPy Concepts

This problem explores the fundamentals of **NumPy**, which is the core library for numerical computing in Python. We will demonstrate several key operations on NumPy arrays.

- **NumPy Arrays**: These are powerful, grid-like data structures that are much more efficient for numerical operations than standard Python lists. All our demonstrations will be on multi-dimensional arrays.
- **Broadcasting**: This is NumPy's powerful mechanism for performing arithmetic operations on arrays of different shapes. The smaller array's values are "broadcast" across the larger array so they can be combined.
- **Dot Product**: A central operation in linear algebra that defines how to multiply matrices. It's different from simple element-wise multiplication.
- **Determinant and Inverse**: These are properties of square matrices. The **determinant** is a scalar value that provides important information about the matrix, while the **inverse** is another matrix that, when multiplied by the original, results in the identity matrix. These are crucial for solving systems of linear equations.

In [6]:
```
print("=== PROBLEM 4: Array Operations and Linear Algebra ===\n")

# --- 1. Array Creation ---
# We create two arrays: a general 3x4 array for most operations,
# and a 3x3 square matrix for linear algebra.
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])

square_matrix = np.array([[4, 7, 2],
                          [2, 6, 0],
                          [1, 2, 5]])
print("--- 1. Initial Arrays ---")
print("General purpose 3x4 array:\n", arr)
print("\nSquare 3x3 matrix for linear algebra:\n", square_matrix)
```

```python
# --- 2. Shape and Resize Operations ---
print("\n\n--- 2. Shape and Resize Operations ---")
print(f"Shape of arr: {arr.shape}")
print(f"Size of arr: {arr.size}")
print(f"Transposed arr:\n{arr.T}")
print(f"Flattened arr: {arr.flatten()}")
print(f"Reshaped to 2x6:\n{arr.reshape(2, 6)}")


# --- 3. Negative Indexing and Slicing Error ---
print("\n\n--- 3. Negative Indexing and Slicing Error ---")
print(f"Using negative indexing to get the last element: {arr[-1, -1]}")

print("\nShowcasing a slicing error:")
print("The following line of code is commented out because it will stop the sc
print("arr[0, 5]  <-- This would cause an IndexError because column 5 does not
# arr[0, 5] # Uncomment this line to see the IndexError


# --- 4. Arithmetic Operations ---
print("\n\n--- 4. Arithmetic Operations ---")
# Broadcasting: adding a scalar to the whole array
print("Broadcasting a scalar (arr + 100):\n", arr + 100)
# Broadcasting: adding a 1D array to a 2D array
broadcast_row = np.array([10, 20, 30, 40])
print("\nBroadcasting a 1D array to the 2D array:\n", arr + broadcast_row)

# Dot Product
arr_a = np.array([[1, 2], [3, 4], [5, 6]]) # Shape: 3x2
arr_b = np.array([[10, 20, 30], [40, 50, 60]]) # Shape: 2x3
print("\nDot product of a 3x2 and a 2x3 matrix:\n", np.dot(arr_a, arr_b))


# --- 5. Linear Algebra ---
print("\n\n--- 5. Linear Algebra ---")
# Determinant
det = np.linalg.det(square_matrix)
print(f"Determinant of the square matrix: {det:.2f}")

# Inverse
# We can only find the inverse if the determinant is non-zero.
if det != 0:
    inverse = np.linalg.inv(square_matrix)
    print("\nInverse of the square matrix:\n", inverse)
else:
    print("\nMatrix is not invertible (determinant is zero).")
```

```
=== PROBLEM 4: Array Operations and Linear Algebra ===

--- 1. Initial Arrays ---
General purpose 3x4 array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

Square 3x3 matrix for linear algebra:
 [[4 7 2]
 [2 6 0]
 [1 2 5]]


--- 2. Shape and Resize Operations ---
Shape of arr: (3, 4)
Size of arr: 12
Transposed arr:
[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]
Flattened arr: [ 1  2  3  4  5  6  7  8  9 10 11 12]
Reshaped to 2x6:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]


--- 3. Negative Indexing and Slicing Error ---
Using negative indexing to get the last element: 12

Showcasing a slicing error:
The following line of code is commented out because it will stop the script.
arr[0, 5]  <-- This would cause an IndexError because column 5 does not exist.


--- 4. Arithmetic Operations ---
Broadcasting a scalar (arr + 100):
 [[101 102 103 104]
 [105 106 107 108]
 [109 110 111 112]]

Broadcasting a 1D array to the 2D array:
 [[11 22 33 44]
 [15 26 37 48]
 [19 30 41 52]]

Dot product of a 3x2 and a 2x3 matrix:
 [[ 90 120 150]
 [190 260 330]
 [290 400 510]]


--- 5. Linear Algebra ---
```

```
Determinant of the square matrix: 46.00

Inverse of the square matrix:
 [[ 0.65217391 -0.67391304 -0.26086957]
 [-0.2173913   0.39130435  0.08695652]
 [-0.04347826 -0.02173913  0.2173913 ]]
```

Solution Methodology for Problem 4

My approach was to systematically demonstrate each required NumPy operation in its own clearly marked section.

1. **Array Creation**: I initialized two separate arrays: a general-purpose 3x4 array for most demonstrations and a 3x3 square matrix specifically for the linear algebra operations, which require a square shape.
2. **Shape Operations**: I showcased the `.shape` and `.size` attributes, and the `.T` (transpose), `.flatten()`, and `.reshape()` methods to cover all the required shape and resize demonstrations.
3. **Indexing and Errors**: I provided a clear example of negative indexing to fetch the last element. To showcase a slicing error, I wrote a line of code that would produce an `IndexError` and explained in a comment why it would fail, keeping it commented out so the entire notebook can run without interruption.
4. **Arithmetic**: I demonstrated broadcasting by adding both a scalar and a 1D array to a 2D array. For the dot product, I created two new matrices with compatible inner dimensions (3x**2** and **2**x3) to correctly perform the operation.
5. **Linear Algebra**: Using the square matrix, I calculated its determinant with `np.linalg.det()`. After confirming the determinant was non-zero, I computed the matrix inverse using `np.linalg.inv()`.

Assignment Completion Summary

This notebook successfully addressed all four problems of the practical assignment, demonstrating key skills in data analysis and numerical computing with Python.

Summary of Tasks Completed:

- **1. Statistical Measures:** We calculated the mean, median, and age-

weighted mean for the income data. The concept and appropriate use of a weighted mean were also explained. Proper handling of `NaN` values was ensured throughout.

- **2. Standardization & Outliers:** The income data was standardized by calculating z-scores, which were added as a new column to the DataFrame. Outliers were then identified and counted based on the standard `|z| > 3` rule.
- **3. Age Binning & Aggregation:** We successfully segmented the dataset into the required age bins using `pd.cut`. The `groupby().agg()` method was then used to efficiently compute the count, mean income, and median score for each group, presenting the results in a clean, tidy DataFrame.
- **4. NumPy Operations:** A comprehensive demonstration of fundamental NumPy operations was provided. This included array shape manipulation, negative indexing, showcasing a slicing error, and performing arithmetic operations like broadcasting and the dot product. Key linear algebra functions like calculating a determinant and matrix inverse were also successfully executed.

Key Learnings:

This assignment provided practical experience in several core data science concepts:

- The critical importance of setting a `random.seed` for creating reproducible analyses.
- How different measures of central tendency (mean, median, weighted mean) can provide different insights into a dataset.
- The utility of z-scores as a universal method for identifying statistical outliers.
- The power and efficiency of using `groupby().agg()` to perform complex data summaries in a single command.

All assignment requirements have been met, and the solutions have been presented in a clean, well-documented, and reproducible format.