**MKTG5883 Exercise 5: Data Visualization**

**Tasks:**

**Exercise 1 - Basic Viz**

1. Prepare the data

# Store the provided data in a string format

data\_string = """Day of Week,Month,Day,Year,Visits

Monday,1-Jun,1,2009,27

Tuesday,2-Jun,2,2009,31

Wednesday,3-Jun,3,2009,38

Thursday,4-Jun,4,2009,38

Friday,5-Jun,5,2009,31

Saturday,6-Jun,6,2009,24

Sunday,7-Jun,7,2009,21

Monday,8-Jun,8,2009,29

Tuesday,9-Jun,9,2009,30

Wednesday,10-Jun,10,2009,22

Thursday,11-Jun,11,2009,24

Friday,12-Jun,12,2009,17

Saturday,13-Jun,13,2009,7

Sunday,14-Jun,14,2009,13

A graph of blue bars

AI-generated content may be incorrect.

A graph with blue and red squares

AI-generated content may be incorrect.

A graph with green line

AI-generated content may be incorrect.

**Plotting Library:** Primarily uses matplotlib.pyplot (imported as plt) functions like plt.bar, plt.boxplot, and plt.plot directly, similar to the style in the image.

**Part (a) - Bar Chart:**

Calculates the *mean* visits per day using df.groupby("Day of Week")["Visits"].mean().

Uses weekly\_visits\_mean.reindex(day\_order) to ensure the bars are plotted Monday to Sunday.

Uses plt.bar() to create the bar chart.

**Part (a) - Box Plot (Matplotlib version):**

Manually prepares the data as a list of arrays, where each array contains the visits for a specific day of the week in the correct order.

Uses plt.boxplot() to create the box plot, passing the prepared data list and the day\_order for labels. Added some basic styling (patch\_artist, medianprops, boxprops).

**Part (b) - Line Plot:**

Uses plt.plot() instead of sns.lineplot. The arguments (x, y, marker, linestyle, color) achieve a similar result.

import pandas as pd

import matplotlib.pyplot as plt

import io

import numpy as np # Needed for boxplot if not using seaborn

# 1. Prepare the data

# Store the provided data in a string format

data\_string = """Day of Week,Month,Day,Year,Visits

Monday,1-Jun,1,2009,27

Tuesday,2-Jun,2,2009,31

Wednesday,3-Jun,3,2009,38

Thursday,4-Jun,4,2009,38

Friday,5-Jun,5,2009,31

Saturday,6-Jun,6,2009,24

Sunday,7-Jun,7,2009,21

Monday,8-Jun,8,2009,29

Tuesday,9-Jun,9,2009,30

Wednesday,10-Jun,10,2009,22

Thursday,11-Jun,11,2009,24

Friday,12-Jun,12,2009,17

Saturday,13-Jun,13,2009,7

Sunday,14-Jun,14,2009,13

"""

# Read the data into a pandas DataFrame

# Using io.StringIO to read the string data as if it were a file

df = pd.read\_csv(io.StringIO(data\_string))

# 2. Data Preprocessing

# Combine the 'Month' column (which contains day-month) and 'Year' column

df['Date\_Str'] = df['Month'] + '-' + df['Year'].astype(str)

# Convert the string date to datetime objects

df['Date'] = pd.to\_datetime(df['Date\_Str'], format='%d-%b-%Y')

# Sort the DataFrame by date to ensure chronological order for time-series plotting

df = df.sort\_values(by='Date')

# Define the order of the days of the week for plotting

day\_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

# Use strip() on column name to remove potential leading/trailing spaces if reading from file

df['Day of Week'] = df['Day of Week'].str.strip()

# Display the first few rows and data types to verify

print("DataFrame Head:")

print(df.head())

print("\nDataFrame Info:")

df.info()

print("\n" + "="\*50 + "\n")

# ----------------------------------------------------------------------------

# a) Chart showing variability/traffic for each day of the week

# ----------------------------------------------------------------------------

# --- Option 1: Bar Chart of Average Visits (Similar to the image provided) ---

print("Generating Plot for Part (a): Bar Chart of Average Visits")

# Group by DayOfWeek and get average visits

# Use observed=False for newer pandas versions if Day of Week was categorical

# For this specific grouping, it might not be necessary yet.

weekly\_visits\_mean = df.groupby("Day of Week")["Visits"].mean()

# Sort days in correct order using reindex

weekly\_visits\_mean = weekly\_visits\_mean.reindex(day\_order)

# Plot bar chart using matplotlib

plt.figure(figsize=(10, 6)) # Set figure size

plt.bar(weekly\_visits\_mean.index, weekly\_visits\_mean.values, color='skyblue') # Create bar chart

plt.xlabel("Day of the Week") # X-axis label

plt.ylabel("Average Number of Visits") # Y-axis label

plt.title("Average Website Visits by Day of the Week") # Chart title

plt.xticks(rotation=45) # Rotate x-axis labels for better readability

plt.grid(axis='y', linestyle='--', alpha=0.7) # Add horizontal grid lines

plt.tight\_layout() # Adjust layout

plt.show() # Display the plot

# --- Option 2: Box Plot showing Variability (More direct answer to 'variability') ---

print("\nGenerating Plot for Part (a): Box Plot of Visits Distribution")

# Prepare data for boxplot: Create a list of visit arrays for each day

visits\_by\_day = [df[df['Day of Week'] == day]['Visits'].values for day in day\_order]

# Plot box plot using matplotlib

plt.figure(figsize=(10, 6))

# Use positions to ensure boxes align with labels correctly

plt.boxplot(visits\_by\_day, labels=day\_order, patch\_artist=True, # patch\_artist fills boxes

            medianprops={'color': 'red', 'linewidth': 2}, # Customize median line

            boxprops={'facecolor': 'lightblue'}) # Customize box color

plt.xlabel("Day of the Week")

plt.ylabel("Number of Visits")

plt.title("Website Visits Variability by Day of the Week (Box Plot)")

plt.xticks(rotation=45) # Matplotlib uses numeric ticks for boxplot, labels map to these

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight\_layout()

plt.show()

print("\n" + "="\*50 + "\n")

# ----------------------------------------------------------------------------

# b) Time-based trends

# ----------------------------------------------------------------------------

print("Generating Plot for Part (b): Time-Based Trend")

# Create the line plot using matplotlib

plt.figure(figsize=(12, 6)) # Set figure size

plt.plot(df['Date'], df['Visits'], marker='o', linestyle='-', color='green') # Plot date vs visits

# Add titles and labels

plt.title('Website Visits Over Time (June 1st - June 14th, 2009)')

plt.xlabel('Date')

plt.ylabel('Number of Visits')

plt.grid(True, linestyle='--', alpha=0.7) # Add grid lines

# Rotate date labels for better readability

plt.xticks(rotation=45)

# Show the plot

plt.tight\_layout() # Adjust layout

plt.show()

# Interpretation for Part (b):

# The line plot shows the number of visits sequentially over the two-week period.

# - We can observe a general pattern: visits tend to be higher during the weekdays and lower during the weekends.

# - There seems to be a peak mid-week in the first week (Wed/Thu).

# - There's a noticeable drop in visits during the second weekend (June 13-14), reaching the lowest point in the dataset on Saturday, June 13th.

# - There might be a slight downward trend from the first week to the second week, particularly noticeable in the weekend lows and the peak weekday visits.

**2. Exercise 2 – Sampling**

**a/ Explain why output distribution is likely fallen into one cluster only.**

The output distribution is likely fallen into one cluster only because when using the permutation means that we are sampling equally. There are 1020 data points and the probability fall into cluster one is 98%. The probabilit to choose sample from another 2 cluster is 1% for each.

import numpy as np

def algo1(points, k=10):

# points would be an array or list representing all servers

perm = np.random.permutation(points) # Randomly shuffle ALL points

return perm[:k] # Select the first k points from the shuffled list

**b/ Write an algorithm in pseudo-code or Python to select the samples.**

**Allocation Logic:**

**Requirement 1 (Min 1 per cluster):** We have 3 clusters. Allocate 1 sample to each cluster initially.

Samples allocated = 3

Remaining samples = k - num\_clusters = 10 - 3 = 7

**Requirement 2 (Proportionality):** Distribute the remaining 7 samples proportionally based on cluster size.

Cluster sizes: C1=1000, C2=10, C3=10. Total = 1020.

Proportions: P1 ≈ 0.9804, P2 ≈ 0.0098, P3 ≈ 0.0098.

Additional samples for C1 ≈ 7 \* 0.9804 ≈ 6.86

Additional samples for C2 ≈ 7 \* 0.0098 ≈ 0.07

Additional samples for C3 ≈ 7 \* 0.0098 ≈ 0.07

**Rounding:** We need whole numbers. Rounding the additional samples gives:

Additional for C1: round(6.86) = 7

Additional for C2: round(0.07) = 0

Additional for C3: round(0.07) = 0

Check sum: 7 + 0 + 0 = 7. This matches the remaining samples.

**Final Allocation:** Combine initial and proportional samples:

Samples from C1 (Large) = 1 (initial) + 7 (proportional) = 8

Samples from C2 (Small) = 1 (initial) + 0 (proportional) = 1

Samples from C3 (Small) = 1 (initial) + 0 (proportional) = 1

Total Samples = 8 + 1 + 1 = 10. This matches k.

import numpy as np

import pandas as pd

from sklearn.cluster import KMeans # Or use known labels if provided

def simulate\_server\_data(n\_total=1020, seed=42):

    """Creates dummy server data with 3 clusters."""

    np.random.seed(seed)

    # Cluster 1 (Large)

    c1 = np.random.randn(1000, 2) \* 0.5 + np.array([2, 2])

    # Cluster 2 (Small)

    c2 = np.random.randn(10, 2) \* 0.2 + np.array([-1, -1])

    # Cluster 3 (Small)

    c3 = np.random.randn(10, 2) \* 0.2 + np.array([-1.5, 1.5])

    points = np.vstack((c1, c2, c3))

    # Create labels (assuming we know them, or use clustering)

    # In a real scenario, you might run KMeans first if labels are unknown

    labels = np.array([0]\*1000 + [1]\*10 + [2]\*10)

    # Shuffle points and labels together

    indices = np.arange(n\_total)

    np.random.shuffle(indices)

    points = points[indices]

    labels = labels[indices]

    df = pd.DataFrame(points, columns=['errors', 'users'])

    df['cluster\_label'] = labels

    # Add an index to track original points if needed

    df['original\_index'] = df.index

    return df

def stratified\_proportional\_sampling(df, label\_column, k=10):

    """

    Performs stratified sampling ensuring at least one sample per stratum

    and proportionally allocating the rest based on stratum size.

    Args:

        df (pd.DataFrame): DataFrame containing the data points and cluster labels.

                           Must include an 'original\_index' column if you want to

                           return indices referring to the original DataFrame.

        label\_column (str): The name of the column containing cluster labels.

        k (int): The total number of samples desired.

    Returns:

        pd.DataFrame: A DataFrame containing the selected samples.

                      OR

        np.ndarray: An array of the original indices of the selected samples.

    """

    clusters = df[label\_column].unique()

    num\_clusters = len(clusters)

    if k < num\_clusters:

        raise ValueError(f"k ({k}) must be at least the number of clusters ({num\_clusters})")

    # Initial allocation: 1 per cluster

    samples\_per\_cluster = {cluster\_id: 1 for cluster\_id in clusters}

    remaining\_samples = k - num\_clusters

    print(f"Initial allocation: {samples\_per\_cluster}")

    print(f"Remaining samples to allocate proportionally: {remaining\_samples}")

    if remaining\_samples > 0:

        cluster\_sizes = df[label\_column].value\_counts()

        total\_population = len(df)

        proportions = cluster\_sizes / total\_population

        # Calculate proportional allocation for remaining samples

        proportional\_alloc = (proportions \* remaining\_samples)

        # Round intelligently - prioritizing larger fractions first helps

        # Or use a more robust library function if available. Simple rounding here:

        proportional\_alloc\_rounded = proportional\_alloc.round().astype(int)

        print(f"Calculated proportional allocations (rounded): \n{proportional\_alloc\_rounded}")

        # --- Adjustment for rounding errors ---

        current\_sum = proportional\_alloc\_rounded.sum()

        diff = remaining\_samples - current\_sum

        print(f"Sum check: current\_sum={current\_sum}, needed={remaining\_samples}, diff={diff}")

        if diff != 0:

             # Adjust the difference, typically by adding/subtracting from largest clusters

             # Find clusters sorted by the fractional part of proportional\_alloc, descending

             # This helps decide where rounding likely had the most impact

             fractional\_order = (proportional\_alloc - np.floor(proportional\_alloc)).sort\_values(ascending=False).index

             for i in range(abs(diff)):

                 idx\_to\_adjust = fractional\_order[i % len(fractional\_order)]

                 proportional\_alloc\_rounded[idx\_to\_adjust] += np.sign(diff)

             print(f"Adjusted proportional allocations: \n{proportional\_alloc\_rounded}")

             # Final check after adjustment

             final\_check\_sum = proportional\_alloc\_rounded.sum()

             if final\_check\_sum != remaining\_samples:

# Fallback if adjustment logic fails (rare but possible with complex rounding)

print(f"Warning: Adjustment failed. Sum is {final\_check\_sum}, needed {remaining\_samples}. Manual override might be needed.")

                 # A simple fallback: just add/remove from the largest cluster

                 largest\_cluster\_id = cluster\_sizes.idxmax()

                 proportional\_alloc\_rounded[largest\_cluster\_id] += (remaining\_samples - final\_check\_sum)

        # Add proportional allocation to initial allocation

        for cluster\_id, count in proportional\_alloc\_rounded.items():

             samples\_per\_cluster[cluster\_id] += count

    print(f"Final allocation per cluster: {samples\_per\_cluster}")

    # --- Perform sampling within each cluster ---

    final\_sample\_indices = []

for cluster\_id, num\_to\_sample in samples\_per\_cluster.items():

cluster\_df = df[df[label\_column] == cluster\_id]

# Ensure we don't try to sample more than available

actual\_sample\_size = min(num\_to\_sample, len(cluster\_df))

if actual\_sample\_size > 0:

# Sample 'original\_index' to refer back to the main df

sampled\_indices = np.random.choice(

cluster\_df['original\_index'],

size=actual\_sample\_size,

replace=False # Sample without replacement

)

final\_sample\_indices.extend(sampled\_indices)

print(f"\nTotal samples selected: {len(final\_sample\_indices)}")

# Return the sampled rows from the original dataframe

return df.loc[final\_sample\_indices]

# Alternatively, return just the indices:

# return np.array(final\_sample\_indices)

# --- Example Usage ---

# 1. Simulate data

server\_df = simulate\_server\_data()

print("Simulated Data Head:")

print(server\_df.head())

print("\nCluster Sizes:")

print(server\_df['cluster\_label'].value\_counts())

print("-" \* 30)

# 2. Run the sampling algorithm

print("Running Stratified Proportional Sampling...\n")

selected\_samples\_df = stratified\_proportional\_sampling(server\_df, 'cluster\_label', k=10)

print("\nSelected Samples Head:")

print(selected\_samples\_df.head())

print("\nSelected Samples Cluster Distribution:")

print(selected\_samples\_df['cluster\_label'].value\_counts())

# You can plot this to visualize

# import matplotlib.pyplot as plt

# plt.figure(figsize=(8, 6))

# plt.scatter(server\_df['errors'], server\_df['users'], c=server\_df['cluster\_label'], alpha=0.2, label='All Points', s=10)

# plt.scatter(selected\_samples\_df['errors'], selected\_samples\_df['users'], c='red', marker='X', s=100, label='Sampled Points')

# plt.title('Server Data and Stratified Sample (k=10)')

# plt.xlabel('Number of Errors')

# plt.ylabel('Number of Users')

# plt.legend()

# plt.grid(True)

# plt.show()

Simulated Data Head:

errors users cluster\_label original\_index

0 1.909760 3.596554 0 0

1 2.112046 2.006296 0 1

2 1.550793 2.245960 0 2

3 1.311165 1.531087 0 3

4 2.014091 1.995441 0 4

Cluster Sizes:

cluster\_label

0 1000

1 10

2 10

Name: count, dtype: int64

------------------------------

Running Stratified Proportional Sampling...

Initial allocation: {np.int64(0): 1, np.int64(1): 1, np.int64(2): 1}

Remaining samples to allocate proportionally: 7

Calculated proportional allocations (rounded):

cluster\_label

0 7

1 0

2 0

Name: count, dtype: int64

Sum check: current\_sum=7, needed=7, diff=0

Final allocation per cluster: {np.int64(0): 8, np.int64(1): 1, np.int64(2): 1}

Total samples selected: 10

Selected Samples Head:

errors users cluster\_label original\_index

638 1.870204 1.248429 0 638

936 1.818780 1.440165 0 936

34 2.097923 1.510814 0 34

131 2.819982 2.371064 0 131

447 2.775250 1.500823 0 447

Selected Samples Cluster Distribution:

cluster\_label

0 8

1 1

2 1

Name: count, dtype: int64