Assignment _3_Xiaoyang Ye

March 19, 2024

1 Assignment 3

1.1 Task 1

[1]:
```
# Import necessary dependencies
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import plotly.offline as pyo
import plotly.graph_objs as go
import plotly.express as px
from plotly.subplots import make_subplots
```

[2]:
```
# Load the data
# I had already upload the data in the jupyter notebook
df = pd.read_csv("turkiye-student-evaluation_generic.csv")
```

1.1.1 1. Data Cleaning & Transformation

1.1 Viewing the data structure and data types.

[3]:
```
print("The first 5 rows of the dataset")
print("------------------------------")
display(df.head(5))

print("Detailed information about the dataset:")
print("----------------------------------------")
display(df.info())
```

```
The first 5 rows of the dataset
------------------------------
```

	instr	class	nb.repeat	attendance	difficulty	Q1	Q2	Q3	Q4	Q5	…	\
0	1	2	1	0	4	3	3	3	3	3	…	
1	1	2	1	1	3	3	3	3	3	3	…	
2	1	2	1	2	4	5	5	5	5	5	…	
3	1	2	1	1	3	3	3	3	3	3	…	
4	1	2	1	0	1	1	1	1	1	1	…	

```
   Q19  Q20  Q21  Q22  Q23  Q24  Q25  Q26  Q27  Q28
0    3    3    3    3    3    3    3    3    3    3
1    3    3    3    3    3    3    3    3    3    3
2    5    5    5    5    5    5    5    5    5    5
3    3    3    3    3    3    3    3    3    3    3
4    1    1    1    1    1    1    1    1    1    1

[5 rows x 33 columns]
Detailed information about the dataset:
---------------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5820 entries, 0 to 5819
Data columns (total 33 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instr       5820 non-null   int64
 1   class       5820 non-null   int64
 2   nb.repeat   5820 non-null   int64
 3   attendance  5820 non-null   int64
 4   difficulty  5820 non-null   int64
 5   Q1          5820 non-null   int64
 6   Q2          5820 non-null   int64
 7   Q3          5820 non-null   int64
 8   Q4          5820 non-null   int64
 9   Q5          5820 non-null   int64
 10  Q6          5820 non-null   int64
 11  Q7          5820 non-null   int64
 12  Q8          5820 non-null   int64
 13  Q9          5820 non-null   int64
 14  Q10         5820 non-null   int64
 15  Q11         5820 non-null   int64
 16  Q12         5820 non-null   int64
 17  Q13         5820 non-null   int64
 18  Q14         5820 non-null   int64
 19  Q15         5820 non-null   int64
 20  Q16         5820 non-null   int64
 21  Q17         5820 non-null   int64
 22  Q18         5820 non-null   int64
 23  Q19         5820 non-null   int64
 24  Q20         5820 non-null   int64
 25  Q21         5820 non-null   int64
 26  Q22         5820 non-null   int64
 27  Q23         5820 non-null   int64
 28  Q24         5820 non-null   int64
 29  Q25         5820 non-null   int64
 30  Q26         5820 non-null   int64
```

```
 31  Q27          5820 non-null   int64
 32  Q28          5820 non-null   int64
dtypes: int64(33)
memory usage: 1.5 MB
```

```
None
```

From the data resource website(https://archive.ics.uci.edu/dataset/262/turkiye+student+evaluation), we can identify the following columns:

Instructor's identifier ("instr") Course code ("class") Number of times the student is taking this course ("repeat") Attendance ("attendance") Level of difficulty of the course as perceived by the student ("difficulty") Answers to all the questions ("Q1", "Q2", ..., "Q28").

Detailed information about the questions can be found on the resource website.

All columns contain numeric data, specifically of type int64.

1.2 Handle missing data.

[4]:
```
missing_values = df.isnull().sum()

display(missing_values)

print(f"There are total {missing_values.sum()} missing values in the dataset.")
```

```
instr         0
class         0
nb.repeat     0
attendance    0
difficulty    0
Q1            0
Q2            0
Q3            0
Q4            0
Q5            0
Q6            0
Q7            0
Q8            0
Q9            0
Q10           0
Q11           0
Q12           0
Q13           0
Q14           0
Q15           0
Q16           0
Q17           0
Q18           0
Q19           0
Q20           0
```

```
Q21           0
Q22           0
Q23           0
Q24           0
Q25           0
Q26           0
Q27           0
Q28           0
dtype: int64
```

```
There are total 0 missing values in the dataset.
```

Ther are no missing values in all columns.

1.3 Address duplicate records Based on the data description, pinpointing duplicate records proves challenging due to the absence of student information. It's difficult to ascertain if multiple instances of the same question were answered by a single student. When all values in a row across all columns are identical, distinguishing duplicate records becomes impossible.

1.4 Correct any inaccuracies or inconsistencies in the data

[276]:
```
# Use the summary report and box plot to check the outliers or inconsistence␣
 ↪values in the data
# (1) Check the mathematical statistics of the numerical data types
df.describe()
```

[276]:

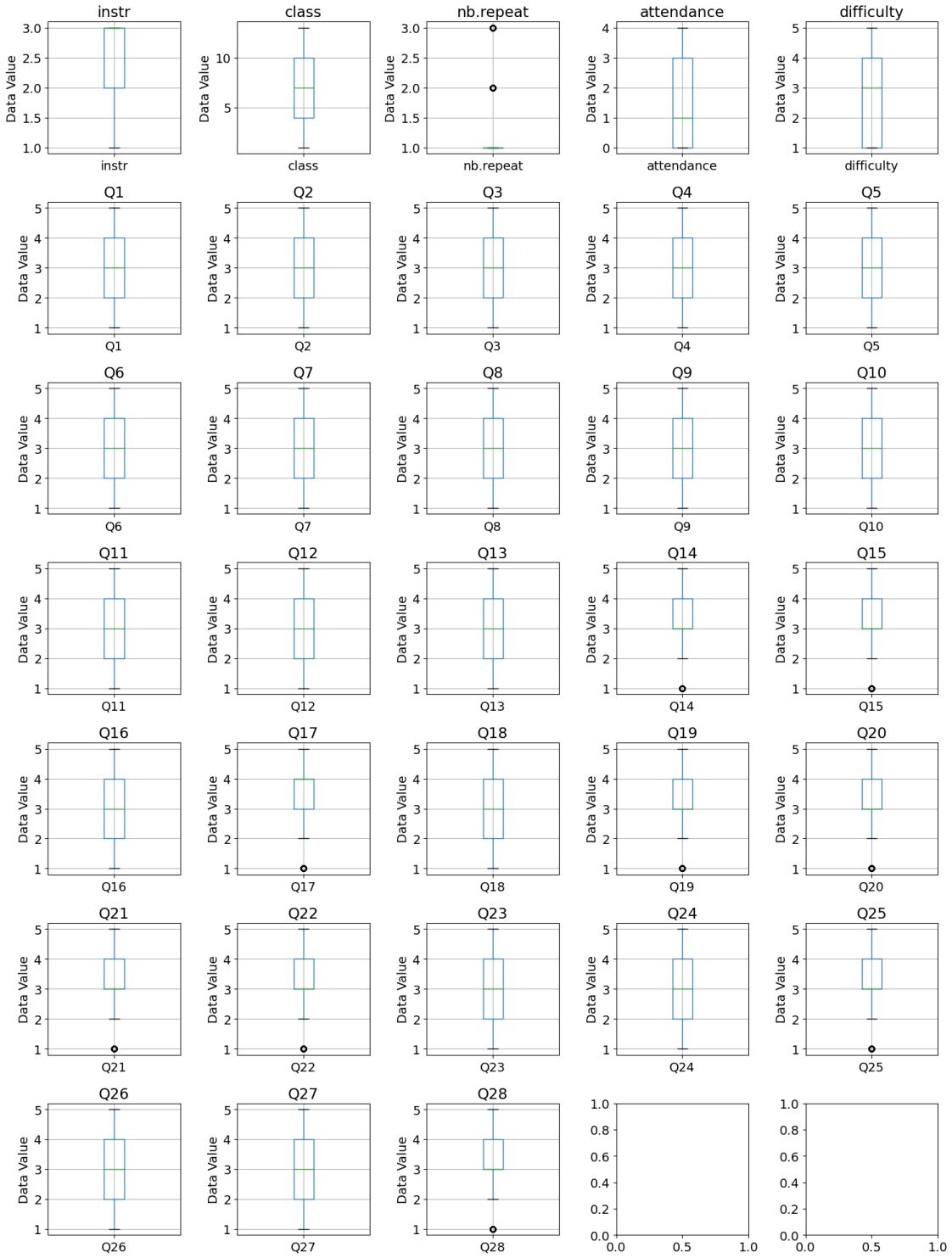
	instr	class	nb.repeat	attendance	difficulty	\
count	5820.000000	5820.000000	5820.000000	5820.000000	5820.000000	
mean	2.485567	7.276289	1.214089	1.675601	2.783505	
std	0.718473	3.688175	0.532376	1.474975	1.348987	
min	1.000000	1.000000	1.000000	0.000000	1.000000	
25%	2.000000	4.000000	1.000000	0.000000	1.000000	
50%	3.000000	7.000000	1.000000	1.000000	3.000000	
75%	3.000000	10.000000	1.000000	3.000000	4.000000	
max	3.000000	13.000000	3.000000	4.000000	5.000000	

	Q1	Q2	Q3	Q4	Q5	…	\
count	5820.000000	5820.000000	5820.000000	5820.000000	5820.000000	…	
mean	2.929897	3.073883	3.178694	3.082474	3.105842	…	
std	1.341077	1.285251	1.253567	1.284594	1.278989	…	
min	1.000000	1.000000	1.000000	1.000000	1.000000	…	
25%	2.000000	2.000000	2.000000	2.000000	2.000000	…	
50%	3.000000	3.000000	3.000000	3.000000	3.000000	…	
75%	4.000000	4.000000	4.000000	4.000000	4.000000	…	
max	5.000000	5.000000	5.000000	5.000000	5.000000	…	

	Q19	Q20	Q21	Q22	Q23	\
count	5820.000000	5820.000000	5820.000000	5820.000000	5820.00000	
mean	3.261684	3.285395	3.307388	3.317526	3.20189	

```
std       1.268442     1.276848     1.269974     1.268358     1.27259
min       1.000000     1.000000     1.000000     1.000000     1.00000
25%       3.000000     3.000000     3.000000     3.000000     2.00000
50%       3.000000     3.000000     3.000000     3.000000     3.00000
75%       4.000000     4.000000     4.000000     4.000000     4.00000
max       5.000000     5.000000     5.000000     5.000000     5.00000

                Q24          Q25          Q26          Q27          Q28
count  5820.000000  5820.000000  5820.000000  5820.000000  5820.000000
mean      3.166838     3.312543     3.222165     3.154811     3.308076
std       1.275909     1.257286     1.270695     1.291872     1.278709
min       1.000000     1.000000     1.000000     1.000000     1.000000
25%       2.000000     3.000000     2.000000     2.000000     3.000000
50%       3.000000     3.000000     3.000000     3.000000     3.000000
75%       4.000000     4.000000     4.000000     4.000000     4.000000
max       5.000000     5.000000     5.000000     5.000000     5.000000

[8 rows x 33 columns]
```

[277]:
```python
# (2) Use the box plot to visualize the ditribution and outliers of the data.
fig, axes = plt.subplots(7, 5, figsize=(15, 20))

# Plot box plots for each column in a subplot
for i, column_name in enumerate(df.columns):
    row_idx = i // 5
    col_idx = i % 5
    df.boxplot(column=column_name, ax=axes[row_idx, col_idx])
    axes[row_idx, col_idx].set_title(column_name)
    axes[row_idx, col_idx].set_ylabel('Data Value')

# Adjust layout
plt.tight_layout()
plt.show()
```



From the box plots, it's evident that several columns have outliers, including nb.repeat, Q14, Q15, Q17, Q19, Q20, Q21, Q22, Q25, and A28.

In the nb.repeat column, some students have taken the course multiple times, with one student

taking it three times and another two times. This aligns with the possibility mentioned in the resource data description that students may take the course multiple times.

In other Q columns, there are outlier values of '1', which could indicate low scores given by a student. Given the possibility of legitimate low scores, I'll retain these original records.

1.4 Data normalization or scaling Based on both the df.describe() summary and the box plots, it's evident that the numeric data exhibit similar scales, as indicated by their mean, minimum, and maximum values. Since there are no categorical data present, there's no need for any data encoding in this context.

1.5 Feature engineering Following the analysis plan, I will do the following fearture engineering: (1)"sum_Q" and "ave_Q": the total sum and average of all question scores. (2)"sum_Q1_12" and "ave_Q1_12": the sum and average of questions Q1 to Q12 associated with the course. (3)"sum_Q13_28" and "ave_Q13_28": the sum of questions Q13 to Q28 associated with the instructor.

Note that questions Q1 to Q28 are all Likert-type, with values ranging from 1 to 5, where higher numbers indicate positive attitudes.

[5]:
```
# (1) Total sum and average of all question scores
df['sum_Q'] = df.iloc[:, 5:33].sum(axis=1)
df['ave_Q'] = df.iloc[:, 5:33].mean(axis=1)

# (2) Sum and average of questions Q1 to Q12 associated with the course
df['sum_Q1_12'] = df.iloc[:, 5:17].sum(axis=1)
df['ave_Q1_12'] = df.iloc[:, 5:17].mean(axis=1)

# (3) Sum of questions Q13 to Q28 associated with the instructor
df['sum_Q13_28'] = df.iloc[:, 17:33].sum(axis=1)
df['ave_Q13_28'] = df.iloc[:, 17:33].mean(axis=1)
```

[6]:
```
#Check the data after engineering
display(df.info())
display(df.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5820 entries, 0 to 5819
Data columns (total 39 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   instr       5820 non-null   int64
 1   class       5820 non-null   int64
 2   nb.repeat   5820 non-null   int64
 3   attendance  5820 non-null   int64
 4   difficulty  5820 non-null   int64
 5   Q1          5820 non-null   int64
 6   Q2          5820 non-null   int64
 7   Q3          5820 non-null   int64
```

```
 8   Q4          5820 non-null   int64
 9   Q5          5820 non-null   int64
 10  Q6          5820 non-null   int64
 11  Q7          5820 non-null   int64
 12  Q8          5820 non-null   int64
 13  Q9          5820 non-null   int64
 14  Q10         5820 non-null   int64
 15  Q11         5820 non-null   int64
 16  Q12         5820 non-null   int64
 17  Q13         5820 non-null   int64
 18  Q14         5820 non-null   int64
 19  Q15         5820 non-null   int64
 20  Q16         5820 non-null   int64
 21  Q17         5820 non-null   int64
 22  Q18         5820 non-null   int64
 23  Q19         5820 non-null   int64
 24  Q20         5820 non-null   int64
 25  Q21         5820 non-null   int64
 26  Q22         5820 non-null   int64
 27  Q23         5820 non-null   int64
 28  Q24         5820 non-null   int64
 29  Q25         5820 non-null   int64
 30  Q26         5820 non-null   int64
 31  Q27         5820 non-null   int64
 32  Q28         5820 non-null   int64
 33  sum_Q       5820 non-null   int64
 34  ave_Q       5820 non-null   float64
 35  sum_Q1_12   5820 non-null   int64
 36  ave_Q1_12   5820 non-null   float64
 37  sum_Q13_28  5820 non-null   int64
 38  ave_Q13_28  5820 non-null   float64
dtypes: float64(3), int64(36)
memory usage: 1.7 MB

None

             instr        class    nb.repeat   attendance   difficulty  \
count  5820.000000  5820.000000  5820.000000  5820.000000  5820.000000
mean      2.485567     7.276289     1.214089     1.675601     2.783505
std       0.718473     3.688175     0.532376     1.474975     1.348987
min       1.000000     1.000000     1.000000     0.000000     1.000000
25%       2.000000     4.000000     1.000000     0.000000     1.000000
50%       3.000000     7.000000     1.000000     1.000000     3.000000
75%       3.000000    10.000000     1.000000     3.000000     4.000000
max       3.000000    13.000000     3.000000     4.000000     5.000000

                Q1           Q2           Q3           Q4           Q5  …  \
count  5820.000000  5820.000000  5820.000000  5820.000000  5820.000000  …
mean      2.929897     3.073883     3.178694     3.082474     3.105842  …
```

```
std       1.341077     1.285251     1.253567     1.284594     1.278989  …
min       1.000000     1.000000     1.000000     1.000000     1.000000  …
25%       2.000000     2.000000     2.000000     2.000000     2.000000  …
50%       3.000000     3.000000     3.000000     3.000000     3.000000  …
75%       4.000000     4.000000     4.000000     4.000000     4.000000  …
max       5.000000     5.000000     5.000000     5.000000     5.000000  …

                Q25          Q26          Q27          Q28        sum_Q  \
count  5820.000000  5820.000000  5820.000000  5820.000000  5820.000000
mean      3.312543     3.222165     3.154811     3.308076    89.212371
std       1.257286     1.270695     1.291872     1.278709    32.426038
min       1.000000     1.000000     1.000000     1.000000    28.000000
25%       3.000000     2.000000     2.000000     3.000000    72.000000
50%       3.000000     3.000000     3.000000     3.000000    86.000000
75%       4.000000     4.000000     4.000000     4.000000   112.000000
max       5.000000     5.000000     5.000000     5.000000   140.000000

              ave_Q    sum_Q1_12    ave_Q1_12   sum_Q13_28   ave_Q13_28
count  5820.000000  5820.000000  5820.000000  5820.000000  5820.000000
mean      3.186156    37.062543     3.088545    52.149828     3.259364
std       1.158073    14.154666     1.179555    19.080006     1.192500
min       1.000000    12.000000     1.000000    16.000000     1.000000
25%       2.571429    27.000000     2.250000    44.000000     2.750000
50%       3.071429    36.000000     3.000000    51.000000     3.187500
75%       4.000000    48.000000     4.000000    64.000000     4.000000
max       5.000000    60.000000     5.000000    80.000000     5.000000

[8 rows x 39 columns]
```

1.1.2 2. Create 15 distinct visualizations using plotly

2.1 Pie chart shows the frequncy in class, instructor and nb.repeat, attendance and difficulty

[8]:
```
# Calculate the frequency of each category in the 'class' column
cl = df['class'].value_counts()

# Calculate the frequency of each category in the 'instr' column
instr = df['instr'].value_counts()

# Calculate the frequency of each category in the 'nb.repeat' column
repeat = df['nb.repeat'].value_counts()

# Calculate the frequency of each category in the 'attendance' column
attendance = df['attendance'].value_counts()

# Calculate the frequency of each category in the 'difficulty' column
diff = df['difficulty'].value_counts()
```

```
# Create subplots for pie charts
specs = [[{'type': 'pie'}, {'type': 'pie'}, {'type': 'pie'}],
         [{'type': 'pie'}, {'type': 'pie'}, None]]

fig = make_subplots(rows=2, cols=3, specs=specs, subplot_titles=("Class",␣
 ↪"Instructor", "Number of Repeat", "Attendance", "Difficulty"))

# Plot pie charts for 'class', 'instr', 'nb.repeat', 'attendance', and␣
 ↪'difficulty' in subplots
fig.add_trace(go.Pie(labels=cl.index, values=cl.values,␣
 ↪name="Class",showlegend=True), row=1, col=1)
fig.add_trace(go.Pie(labels=instr.index, values=instr.values,␣
 ↪name="Instructor",showlegend=True), row=1, col=2)
fig.add_trace(go.Pie(labels=repeat.index, values=repeat.values, name="Number of␣
 ↪Repeat",showlegend=True), row=1, col=3)
fig.add_trace(go.Pie(labels=attendance.index, values=attendance.values,␣
 ↪name="Attendance",showlegend=True), row=2, col=1)
fig.add_trace(go.Pie(labels=diff.index, values=diff.values,␣
 ↪name="Difficulty",showlegend=True), row=2, col=2)

# Adjust layout
fig.update_layout(height=800, width=1000, title="Figure 1 Pie charts for first␣
 ↪five categories")

# Show the plot
fig.show()
```

From Figure 1, the frequency distribution of values in the first five columns of the dataset can be observed.

In the 'class' column, Course 3, Course 13, and Course 5 emerge as the top three courses with the highest counts in the dataset.

Regarding the 'Instructor' column, Instructor 3 appears to be the most frequent.

In the 'Number of Repeat' column, instances of one-time repetition constitute approximately 84.3% of the total counts.

For the 'Attendance' column, the code "0" signifies the lowest level of attendance, which is the most frequent, followed by codes '3', '4', and '2'. While there is no explicit description provided for the codes 0, 1, 2, 3, and 4, it is plausible that '0' indicates either complete absence from class or minimal attendance.

In the 'Difficulty' column, the most common difficulty level is '3', which represents the median difficulty level. Conversely, the most challenging difficulty level, represented by code '5', accounts for approximately 11.2% of instances, whereas the least challenging difficulty level, code '2', constitutes 9.43% of instances.

Next, I will delve deeper into the class and instructor data to analyze how instructors are distributed across different courses.

2.2 Bar plot shows the count of Class and instrutor

[9]:
```
# Group data by class and instructor, and calculate counts
count_instr_class = df.groupby(['class', 'instr']).size().
 ↪reset_index(name='count')

# Create a bar plot
fig = go.Figure()

# Define a color scale or a list of colors
colors = px.colors.qualitative.Set2

# Add trace for count of instructors with class as color
for i, instructor in enumerate(count_instr_class['instr'].unique()):
    instructor_data = count_instr_class[count_instr_class['instr'] ==␣
 ↪instructor]
    fig.add_trace(go.Bar(x=instructor_data['class'],
                         y=instructor_data['count'],
                         name=f'Instructor {instructor}',
                         hovertemplate='Class: %{x}<br>Count:␣
 ↪%{y}<br>Instructor: %{name}',
                         textposition='auto',
                         marker_color=colors[i % len(colors)]))

# Update layout
fig.update_layout(title='Figure 2 Bar chart shows the count of class by␣
 ↪instructor',
                  xaxis_title='Course Code',
                  yaxis_title='Count',
                  xaxis=dict(tickmode='array',
                             tickvals=count_instr_class['class'].unique(),
                             ticktext=count_instr_class['class'].unique()),
                  barmode='stack')

# Show plot
fig.show()
```

First, we can use bar plots to visualize the count of question records for each course and also the count of instructors in each course.

From Figure 2, we can observe that the top 3 courses with the largest number of students who answered the questions are Course 3, Course 13, and Course 5. Among them, Course 3, 4, 5, 8, 9, and 12 are taught by Instructor 3, while Instructor 1 teaches Course 2, 7, and 10. Instructor 2 teaches Course 1, 6, and 11, and Course 13 is taken by both Instructor 2 and Instructor 3.

We note that Instructor 3 is responsible for teaching 6 classes, Instructor 2 teaches 4 classes, and Instructor 1 teaches 3 classes.

2.3 Density count plot shows the count distribution of instructors and class

[10]:
```
# Create density count plot
fig = px.density_contour(df, x='class', y='instr', title='Figure 3 Contour plot␣
 ↪of class and instructor')

# Define the tick values and the text for the x-axis
tickvals = list(range(1, 14))
ticktext = [str(i) for i in tickvals]  # Convert tick values to strings

# Update the figure layout
fig.update_layout(xaxis_title='Class',
                  yaxis_title='Instructor',
                  xaxis=dict(tickmode='array',
                             tickvals=tickvals, ticktext=ticktext))

# Show the plot
fig.show()
```

To better visualize the density of student records across instructors and classes, it's evident that Class 3 and Class 13 taught by Instructor 3 have the largest group of records, while Class 1 instructed by Instructor 2 has the fewest. This observation suggests that the smaller sample size of records might introduce more bias when evaluating both the class and instructors.

2.4 Sunburst visualizes the attendance distribution across classes and instructors.

[11]:
```
# Create a sunburst plot
fig = px.sunburst(df, path=['class', 'instr', 'attendance'], values='ave_Q',
                  title='Figure 4 Sunburst plot of class, instructor, and␣
 ↪attendance')

#show plot
fig.show()
```

From Figure 4, it's evident that attendance levels 0 and 3 are predominant across classes and instructors, with level 1 being the least frequent. Notably, classes 3, 13, and 9, instructed by instructor 3, exhibit a high proportion of level 0 attendance. Conversely, attendance levels across other classes appear more similar.

2.5 Box plot shows the average score for each course and instructors.

[12]:
```
# Create the box plot
fig = px.box(df, x='class', y='ave_Q', color='instr',labels={'class': 'Course␣
 ↪Code', 'ave_Q': 'Average Score', 'instr': 'Instructor'})

# Update layout
fig.update_layout(title='Figure 5 Box plot shows distribution of the average␣
 ↪score of the total questions for Course by Instructor',
                  xaxis=dict(tickmode='array',
                  tickvals=count_instr_class['class'],
                  ticktext=count_instr_class['class']))
```

```
# Show the plot
fig.show()
```

From Figure 5, it's evident that course 2 (taught by instructor 1), course 1 (taught by instructor 2), and course 10 (also taught by instructor 1) have the highest median average scores.

Instructor 2 teaches four classes, with average scores ranging closely between 3 and 4. On the other hand, instructor 1 teaches three classes, with courses 2 and 10 showing higher average performance scores, while course 7 has the lowest at approximately 3.036. This variation might be linked to the difficulty of the course, warranting further investigation. Despite teaching the most number of classes, instructor 3 has the lowest median average score across all questions, ranging from 3 to 3.2. In course 13, taught by either Instructor 2 or Instructor 3, the median of the average score across all 28 questions is lower for Instructor 3 compared to Instructor 2.

Notably, instructor 2's four courses exhibit multiple outliers with lower average scores, indicating extreme evaluations by some students that don't reflect the overall course performance accurately.

2.6 Heat map shows the distribution of average scores of questions in each course.

[285]:
```
# Create a new DataFrame with the class and question columns
fig6_data = df[['class'] + [f'Q{i}' for i in range(1, 29)]]

# Melt the DataFrame to create a long format
melted_fig6 = pd.melt(fig6_data, id_vars='class', var_name='Question',␣
 ↪value_name='Value')

# Display the melted_fig5
#display(melted_fig5.head())

# Sort the 'Question' columns
melted_fig6['Question'] = melted_fig6['Question'].str.replace('Q', '').
 ↪astype(int)

# Create the heatmap, showing the average score of each question and each␣
 ↪course,
# The value of the mean of scores will display when clicked on the cell.
fig = px.imshow(melted_fig6.pivot_table(index='class', columns='Question',␣
 ↪values='Value', aggfunc='mean'),
                color_continuous_scale='pubu')

# Customize the layout
fig.update_layout(
    title='Figure 6 Heatmap of Questions and Courses',
    xaxis_title='Questions',
    yaxis_title='Course Code',
    xaxis_tickvals=list(range(1, 29)),
    xaxis_ticktext=[f'Q{i}' for i in range(1, 29)],
    yaxis_tickvals=list(range(1, 14)),
```

```
    yaxis_ticktext=[f'Course {i}' for i in range(1, 14)])

# Show the plot
fig.show()
```

From Figure 6, we observe the overall average scores of each question in each course. By interpreting the color density on the map, we notice that courses 1, 2, and 10 exhibit the highest average scores, depicted by darker shades. This observation aligns with Figure 5.

The highest average score among all questions is found in Q17, which pertains to "The Instructor arrived on time for classes" in course 8. Despite not having the highest overall median of average scores across all 28 questions in Figure 5, it appears that other questions in course 8, such as Q1 "The semester course content, teaching method, and evaluation system were provided at the start" and Q8 "The quizzes, assignments, projects, and exams contributed to helping the learning", have relatively lower scores, contributing to the lower average score of course 8.

In Course 1, the average scores of all questions are relatively high and similar. Notably, Q17 (related to on-time class attendance) and Q28 (related to how the instructor treated all students) exhibit the highest average scores (approximately 3.574), while Q1 (related to the content, teaching method, and evaluation at the start) has the lowest score (3.172).

In course 2, the overall scores of all questions are similarly high, consistent with Figure 5. Q14 (related to how well the instructor prepares for the class), Q15 (related to taught and the announced lesson plan), and Q17 (related to on-time class attendance) have the highest scores, with values of 3.707, 3.679, and 3.714 respectively. Conversely, Q1 (related to the content, teaching method, and evaluation) has the lowest score at 3.421.

The pattern continues similarly for courses 3 through 13, where Q17 tends to have the highest score, while Q1 often has the lowest. Across all questions, students express the highest satisfaction with Q17, related to the instructor's on-time class attendance, while they exhibit the least satisfaction with Q1 and Q12, which pertain to content, teaching method, and evaluation, and helping students look at life and the world with a new perspective, respectively.

2.7 Tree map show the realtionship between the difficulty and the average scores of all questions.

[286]:
```
# Create the treemap
fig = px.treemap(df, path=['class', 'difficulty'], values='ave_Q',
                 color='ave_Q', color_continuous_scale='fall')

# Customize the layout
fig.update_layout(
    title='Figure 7 Treemap showing Difficulty and Average Score colored by␣
 ↪average score in each course',
    coloraxis_colorbar=dict(title='Average Score')
)

# Show the plot
fig.show()
```

From Figure 7, overall Course 3 and Course 13 have the highest count of records in the dataset.

In Course 3, the most frequent difficulty level reported by students is Level 1 (easy), with an average score of approximately 3.35 for all questions. Interestingly, Level 2 difficulty has the highest average score of all questions, approximately 3.76. However, students perceive Level 5 difficulty, with an average score of about 3.4, suggesting the discrepancy between perceived difficuty and actual scores given by the student.

In Course 13, the majority of students perceived the difficulty as Level 1, with an overall average score of 3.36. Notably, Level 3 difficulty yielded the highest average score of all questions, approximately 3.63.

Courses 1, 2, 6, 8, 9, 11, 10, and 12 predominantly reported Level 3 difficulty as most frequent, while Courses 5 and 7 had Level 4 difficulty as the most frequent with the high aveage score of questions.

2.8 Scatter plot shows the relationship between the questions associated with the course (Q1 to Q12) and the questions associated with the instructors (Q13 to Q28).

[287]:
```
# Create scatter plot
fig = px.scatter(df, x="ave_Q1_12", y="ave_Q13_28", size="class",symbol="class")

# Update layout properties
fig.update_layout(
    title="Figure 8 Scatter plot shows the relationship of the aveage score of␣
 ↪Q1-Q12 and Q13-Q28",
    legend=dict(title="Class", bgcolor='lightgrey', bordercolor='black'),  #␣
 ↪Customize legend
    width=1000,
    height=800,
    coloraxis_colorbar=dict(title="Class")) # Set colorbar title

# Show the plot
fig.show()
```

From Figure 8, which illustrates the relationship between the questions associated with the course (Q1 to Q12) and the questions associated with the instructors (Q13 to Q28), each data point represents a record evaluated by different students. Fluctuations along the diagonal suggest that the average scores for evaluating the course and instructor are similar. Conversely, data points in the upper left quadrant suggest that while the average scores for the course are low, the scores for the instructors are high, and vice versa for data points in the lower right quadrant.

The scatter plot reveals that there are approximately 5 data points for class 5 and 6 data points for class 13 in the upper left quadrant, indicating that 5 students evaluate the instructors highly but the course poorly, while 6 students do the same for class 13. Conversely, there are few data points in the lower right quadrant, where students rate the course highly but the instructors poorly, notably observed for class 10 and class 9 with 1 and 4 students respectively. These "outliers" may suggest a discrepancy in the evaluations between the course and instructor among a minority of students.

Overall, the majority of students evaluate both the course and instructor similarly, with data points concentrated around the center indicating that most students perceive both aspects at a moderate level.

I will deep into the aveage scores in either course or instructor.

2.9. Density count plot visualizes the distribution of average scores for Q1 to Q12 and Q13 to Q28.

[288]:
```
# Create the density count plot with the default color palette
fig = px.density_contour(df, x="ave_Q1_12", y="ave_Q13_28",
                         marginal_x="histogram", marginal_y="histogram")

# Update the figure layout
fig.update_layout(title="Figure 10 Density contour plot of Q1-12 vs Q13-28",
                  xaxis_title='Average score of Q1 to Q12',
                  yaxis_title='Average score of Q13 to Q28',
                  plot_bgcolor='white')
# Set the color of the axis lines to black
fig.update_xaxes(linecolor='black')
fig.update_yaxes(linecolor='black')

# Show figure
fig.show()
```

Before delving into further analysis, let's visualize the overall distribution of average scores for questions Q1 to Q12 and Q13 to Q28 using density plots. Figure 9 illustrates that the average scores for both sets of questions are predominantly distributed around 3 and 4, with fewer counts around score 2. Notably, there are some inconsistencies in the upper density contour around 3, suggesting that some students rated the course lower while rating the instructors higher. Further exploration into each type of question within classes or instructors will be conducted.

2.10 Histogram shows the frequency distribution of average scores for questions associated with class in each class.

[13]:
```
# Sort the 'class' column to make the animation_frame in order
df_sort_class = df.sort_values('class', ascending=True)

# Create the animated histogram
fig = px.histogram(df_sort_class, x='ave_Q1_12', color='class',␣
 ↪animation_frame='class',opacity=0.7, # Set opacity of bars
                   title='Figure 10 Histogram shows the frequent average scores␣
 ↪of questions assocaited with course in each class')

# Update layout
fig.update_layout( xaxis_title='Average Score of Q1-12',
    yaxis_title='Frequency',
    hovermode='closest',  # Show closest data on hover
    plot_bgcolor='white',  # Set plot background color
    bargap=0.1)  # Set gap between bars

# Update y-axis range
fig.update_yaxes(range=[0, 250])
```

```
# Show the plot
fig.show()
```

From the interactive histogram, we observe various patterns in the frequency distribution of average scores associated with each course: #### Course 1: The most frequent average scores range from 2.75 to 3.24, with 81 counts, and from 3.75 to 4.24, with 79 counts. The least frequent range is between 1.25 and 1.74, with only 2 counts. #### Course 2: The range from 4.75 to 5.24 has the highest frequency, with 41 counts. Scores above 3 have the most counts, indicating that students highly evaluate this course. #### Course 3: Shows peculiar average scores, with the range from 0.9 to 1.09 having 132 counts, followed by 1.9 to 2.09 with 86 counts, and 2.9 to 3.09 with 236 counts, which is the highest. This suggests fluctuating evaluations, with most students providing middle values but others giving either high or very low evaluations. #### Course 4: The most frequent range is from 2.75 to 3.24, with 56 counts. #### Course 5: The most frequent ranges are 2.9 to 3.09 with 130 counts and 3.9 to 4.09 with 115 counts, indicating overall positive evaluations. #### Courses 6, 8, 9, 11, and 13: Show similar patterns to course 5. #### Course 7: Shows high frequency in the range from 3 to 3.49 with 49 counts, 1 to 1.49 with 24 counts, and 4 to 4.49 with 26 counts. Overall, most students provide mid-level evaluations, with some giving very low or high evaluations. #### Course 10: Shows overall high frequency in the higher score ranges. #### Course 12: Has fewer student records compared to other courses, with a wider variation. The most frequent average scores range from 2.5 to 4.5.

2.11 Area plot depicts the trend of two sets of questions for instructor.

[20]:
```
# Create the area plot
fig = px.area(df, x='instr', y=['ave_Q1_12','ave_Q13_28'],
              title='Figure 11 Area plot of average scores by instructor',
              labels={'value': 'Average Score', 'variable': 'Score Type'})

# Update layout
fig.update_layout(xaxis_title='Instructor', yaxis_title='Average Score')

# Show the plot
fig.show()
```

From the area plot, it can be observed that the overall trend of the average score of all questions indicates similarity between instructor 1 and instructor 2, with comparable average scores and trends in both sets of questions. However, instructor 3 exhibits the lowest average scores, particularly in the average score of questions associated with instructors.

2.12 Histogram displays the frequency distribution of average scores for questions associated with instructors, color-coded by class.

[14]:
```
# Sort the 'instr' column to make the animation_frame in order
df_sort_instr = df.sort_values('instr', ascending=True)

# Create the animated histogram
fig = px.histogram(df_sort_instr, x='ave_Q13_28', color='class',␣
 ↪animation_frame='instr',opacity=0.7, # Set opacity of bars
```

```
                    title='Figure 12 Histogram shows the frequent average scores␣
 ↪of questions assocaited with instructor in each instructor color by class')

# Update layout
fig.update_layout( xaxis_title='Average Score of Q12-28',
    yaxis_title='Frequency',
    hovermode='closest',  # Show closest data on hover
    plot_bgcolor='white',  # Set plot background color
    bargap=0.1)  # Set gap between bars

# Update y-axis range
fig.update_yaxes(range=[0,750])

# Show the plot
fig.show()
```

From Figure 12, it's evident that for Instructor 1, the average scores of questions associated with instructors are most frequent in the higher levels of the score range, with 3.9-4.09 being the most frequent range with 172 counts. Similarly, for each class, the distribution of average scores is quite similar.

For Instructor 2, students evaluate the instructor overall with high scores, with the range 3.0-4.09 having the most count of 378. Again, the distribution of average scores across classes is similar.

In the case of Instructor 3, the most frequent range is 2.95-3.04 with 737 counts, followed by 3.95-4.04 with 536 counts, and 0.95-1.04 with 432 counts. However, it's notable that there is also a considerable number of students giving low scores, especially in the range below 2, indicating a mixed perception of Instructor 3. Interestingly, the students giving low scores are mostly from Class 13, suggesting a significant portion of students in that class have a negative attitude towards Instructor 3.

2.13 Violin plot of average scores (Q13-28) by instructor and class

[27]:
```
# Create the violin plot
fig = px.violin(df, x='instr', y='ave_Q13_28', color='class', box=True,
                labels={'instr': 'Instructor', 'ave_Q13_28': 'Average Scores␣
 ↪(Q13-28)', 'class': 'Class'})

# Update the layout
fig.update_layout(title='Figure 13 Violin plot of average scores (Q13-28) by␣
 ↪instructor and class',
                  xaxis_title='Instructor', yaxis_title='Average Scores␣
 ↪(Q13-28)')

# Figure show
fig.show()
```

From the violin plot, it can be observed that the average scores of questions associated with instructors vary across different classes. For instance, in Instructor 1, who teaches classes 2, 7, and 10, the distribution of average scores for questions associated with instructors in class 2 is relatively equal

across the range of 2.5 to 5, while in classes 7 and 10, the average scores are more concentrated around scores 3 and 4.

Instructor 2, who teaches classes 1, 6, 11, and 13, shows a higher frequency of scores 3 and 4 in classes 1, 6, and 11, while in class 13, the average scores are evenly distributed between 3 and 5.

Instructor 3, who teaches the remaining classes, exhibits a similar distribution with more counts around scores 3 and 4 in classes 4, 5, 8, and 9, while in class 3, there is a higher count in average scores of 3, and in class 12, the distribution ranges from 2 to 3.

Class 13, taught by either Instructor 2 or 3, shows an overall higher score distribution, while Instructor 1 has a median score distribution around 3 and a lower score distribution around 1.

2.14 3D Plot of Average Scores and Difficulty by Instructor

[45]:
```
# Create the 3D scatter plot
fig = px.scatter_3d(df, x='ave_Q1_12', y='ave_Q13_28', z='difficulty',
                    color = 'instr',
                    color_continuous_scale='purpor',
                    hover_data=['difficulty'])

# Update the layout
fig.update_layout(title='Figure 14 3D plot of average scores and difficulty by␣
 ↪instructor',
                  scene=dict(xaxis_title='Average Scores (Q1-12)',
                             yaxis_title='Average Scores (Q13-28)',
                             zaxis_title='Difficulty'))

# Update the marker size
fig.update_traces(marker=dict(size=3))

# Show the figure
fig.show()
```

To delve deeper into the relationship between the average scores of the two question sets and the course difficulty, we utilized a 3D scatter plot. Upon examining the scatter plot, it becomes evident that there is no discernible correlation between course difficulty and average score levels. Scores across all difficulty levels exhibit a varied distribution, suggesting that difficulty alone may not significantly influence the average scores. Further visualization and analysis will be conducted to explore potential relationships.

2. 15 Bubble chart of class and instructor with average scores and difficulty

[49]:
```
# Create Scatter plot
fig = px.scatter(df, x='class', y='instr', size='difficulty', color='ave_Q',
                 hover_name='difficulty', size_max=30,
                 labels={'class': 'Class', 'instr': 'Instructor', 'ave_Q':␣
 ↪'Average Scores (Q1-28)',
                         'difficulty': 'Difficulty'})

# Update the layout
```

```
fig.update_layout(title='Figure 15 Bubble chart of class and instructor with␣
 ↪average scores and difficulty',
                  xaxis_title='Class', yaxis_title='Instructor')

# Show the figure
fig.show()
```

From Figure 15, we observe distinct patterns in the relationship between instructor, difficulty level, and average scores. Instructor 1 tends to present higher and middle difficulty levels, with Class 7 exhibiting notably high difficulty (level 5) and lower average scores. Across classes taught by Instructor 1, there is a clear trend: higher difficulty corresponds to lower average scores.

For Instructor 2, difficulty ranges between 3 and 4, with average scores generally aligning in the 3 to 4 range. Notably, difficulty level 1 is associated with an average score of 4. Classes instructed by Instructor 2 display moderate to high difficulty evaluations, accompanied by relatively high average scores.

Instructor 3's classes display a broader range of difficulty, spanning from 1 to 4. Interestingly, there is no clear trend observed between difficulty level and average scores, as higher difficulty levels do not consistently correlate with lower average scores.

1.1.3 3.Plot Correlation matrix of numerical features and write down your insights

[54]:
```
# Calculate the correlation matrix
df_corr = df.corr()

# Create the figure
fig = go.Figure(go.Heatmap(z=corr.values,
                    x=corr.columns,
                    y=corr.columns,
                    colorscale='rdbu'))

# Update the layout
fig.update_layout(title='Correlation matrix of numerical features',
                  xaxis_title='Features',
                  yaxis_title='Features',
                  height=800)

# Show the figure
fig.show()
```

From the correlation matrix, there are no significant correlations observed between the question scores and the instructor, class, number of repeats, attendance, and difficulty. Similarly, the correlations among the instructor, class, number of repeats, attendance, and difficulty are also low.

However, the correlations between each pair of questions are high, with correlation coefficients greater than 0.8 except for question 17. This weaker correlation for question 17 may be attributed to all students consistently giving high scores for it across all classes, thereby reducing the strength of correlation observed.

1.2 Task 2

[124]:
```
# Load the data
# I had already upload the data in the jupyter notebook
df2_1 = pd.read_csv("training.csv")
df2_2 = pd.read_csv("testing.csv")
```

[125]:
```
# Visualize the data structure to determine how to merge these two dataset
display(df2_1.head())
display(df2_2.head())
```

```
       class  BrdIndx  Area  Round  Bright  Compact  ShpIndx  Mean_G  Mean_R  \
0        car     1.27    91   0.97  231.38     1.39     1.47  207.92  241.74
1  concrete      2.36   241   1.56  216.15     2.46     2.51  187.85  229.39
2  concrete      2.12   266   1.47  232.18     2.07     2.21  206.54  244.22
3  concrete      2.42   399   1.28  230.40     2.49     2.73  204.60  243.27
4  concrete      2.15   944   1.73  193.18     2.28     4.10  165.98  205.55

   Mean_NIR  …  SD_NIR_140  LW_140  GLCM1_140  Rect_140  GLCM2_140  \
0    244.48  …       26.18    2.00       0.50      0.85       6.29
1    231.20  …       22.29    2.25       0.79      0.55       8.42
2    245.79  …       15.59    2.19       0.76      0.74       7.24
3    243.32  …       13.51    3.34       0.82      0.74       7.44
4    208.00  …       15.65   50.08       0.85      0.49       8.15

   Dens_140  Assym_140  NDVI_140  BordLngth_140  GLCM3_140
0      1.67       0.70     -0.08             56    3806.36
1      1.38       0.81     -0.09           1746    1450.14
2      1.68       0.81     -0.07            566    1094.04
3      1.36       0.92     -0.09           1178    1125.38
4      0.23       1.00     -0.08           6232    1146.38

[5 rows x 148 columns]
```

```
       class  BrdIndx  Area  Round  Bright  Compact  ShpIndx  Mean_G  Mean_R  \
0  concrete      1.32   131   0.81  222.74     1.66     2.18  192.94  235.11
1    shadow      1.59   864   0.94   47.56     1.41     1.87   36.82   48.78
2    shadow      1.41   409   1.00   51.38     1.37     1.53   41.72   51.96
3      tree      2.58   187   1.91   70.08     3.41     3.11   93.13   55.20
4   asphalt      2.60   116   2.05   89.57     3.06     3.02   73.17   94.89

   Mean_NIR  …  SD_NIR_140  LW_140  GLCM1_140  Rect_140  GLCM2_140  \
0    240.15  …       31.15    5.04       0.80      0.58       8.56
1     57.09  …       12.01    3.70       0.52      0.96       7.01
2     60.48  …       18.75    3.09       0.90      0.63       8.32
3     61.92  …       27.67    6.33       0.89      0.70       8.56
4    100.64  …       32.05    1.01       0.83      0.75       8.62

   Dens_140  Assym_140  NDVI_140  BordLngth_140  GLCM3_140
```

```
0      0.82       0.98     -0.10           1512    1287.52
1      1.69       0.86     -0.14            196    2659.74
2      1.38       0.84      0.10           1198     720.38
3      1.10       0.96      0.20            524     891.36
4      2.08       0.08     -0.10            496    1194.76

[5 rows x 148 columns]
```

[126]:
```
# Concatenate the dataframes vertically
df2 = pd.concat([df2_1, df2_2])

# Check the type of the new data frame
display(df2.head())
display(df2.info())
```

```
       class  BrdIndx  Area  Round  Bright  Compact  ShpIndx  Mean_G  Mean_R  \
0        car     1.27    91   0.97  231.38     1.39     1.47  207.92  241.74
1  concrete      2.36   241   1.56  216.15     2.46     2.51  187.85  229.39
2  concrete      2.12   266   1.47  232.18     2.07     2.21  206.54  244.22
3  concrete      2.42   399   1.28  230.40     2.49     2.73  204.60  243.27
4  concrete      2.15   944   1.73  193.18     2.28     4.10  165.98  205.55

   Mean_NIR  …  SD_NIR_140  LW_140  GLCM1_140  Rect_140  GLCM2_140  \
0    244.48  …       26.18    2.00       0.50      0.85       6.29
1    231.20  …       22.29    2.25       0.79      0.55       8.42
2    245.79  …       15.59    2.19       0.76      0.74       7.24
3    243.32  …       13.51    3.34       0.82      0.74       7.44
4    208.00  …       15.65   50.08       0.85      0.49       8.15

   Dens_140  Assym_140  NDVI_140  BordLngth_140  GLCM3_140
0      1.67       0.70     -0.08             56    3806.36
1      1.38       0.81     -0.09           1746    1450.14
2      1.68       0.81     -0.07            566    1094.04
3      1.36       0.92     -0.09           1178    1125.38
4      0.23       1.00     -0.08           6232    1146.38

[5 rows x 148 columns]
<class 'pandas.core.frame.DataFrame'>
Index: 675 entries, 0 to 506
Columns: 148 entries, class to GLCM3_140
dtypes: float64(133), int64(14), object(1)
memory usage: 785.7+ KB
None
```

1.2.1 1.Data Cleaning & Transformation

1.1 Handle missing data.

[127]:
```
# 1.1.1 Visualize the missing values
missing_values_2 = df2.isnull().sum()
display(missing_values_2)
print(f"There are total {missing_values_2.sum()} missing values in the dataset.
  ↪")
```

```
class            0
BrdIndx          0
Area             0
Round            0
Bright           0
                ..
Dens_140         0
Assym_140        0
NDVI_140         0
BordLngth_140    0
GLCM3_140        0
Length: 148, dtype: int64
```

```
There are total 0 missing values in the dataset.
```

1.2 Address duplicate records

[128]:
```
# I define that all values in all columns are the same as the duplicated records
all_duplicates_2 = df2.duplicated(keep=False)

# Display rows that are duplicates
all_duplicates_df_2 = df2[all_duplicates_2]
print("Duplicate rows where all values in all columns are the same")
display(all_duplicates_df_2)
```

```
Duplicate rows where all values in all columns are the same
```

	class	BrdIndx	Area	Round	Bright	Compact	ShpIndx	Mean_G	Mean_R	\
152	tree	1.26	216	1.03	118.06	1.39	1.5	177.24	79.63	
455	tree	1.26	216	1.03	118.06	1.39	1.5	177.24	79.63	

	Mean_NIR	…	SD_NIR_140	LW_140	GLCM1_140	Rect_140	GLCM2_140	\
152	97.32	…	45.8	1.41	0.92	0.49	8.89	
455	97.32	…	45.8	1.41	0.92	0.49	8.89	

	Dens_140	Assym_140	NDVI_140	BordLngth_140	GLCM3_140
152	1.5	0.52	0.15	690	599.36
455	1.5	0.52	0.15	690	599.36

```
[2 rows x 148 columns]
```

There are two rows with identical values in all columns. I will remove one of them and keep one for further analysis.

[129]:
```
# Remove duplicate rows keeping only one instance
df2_nd = df2.drop_duplicates(keep='first')

# Check the dataframe after removing duplicate rows
display(df2_nd.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 674 entries, 0 to 506
Columns: 148 entries, class to GLCM3_140
dtypes: float64(133), int64(14), object(1)
memory usage: 784.6+ KB
```

```
None
```

1.3 Correct any inaccuracies or inconsistencies in the data

(1) Object data

[130]:
```
#First check the unique values in the object-type columns to determine the␣
 ↪appropriate actions

# Get the list of object-type columns
object_col_df2 = df2_nd.select_dtypes(include=['object']).columns
#display(len(object_col_df2))

# Display unique values for each object-type column
for column in object_col_df2:
    print(f"Unique values in column '{column}':")
    display(df2_nd[column].unique())
```

```
Unique values in column 'class':
```

```
array(['car ', 'concrete ', 'tree ', 'building ', 'asphalt ', 'grass ',
       'shadow ', 'soil ', 'pool '], dtype=object)
```

No typos were found in the unique object.

(2) Numerical data There are 148 columns, making it impractical to use pair plots or box plots to visualize the distribution of numerical data. Instead, I'll use the describe() function to examine any inconsistencies by checking the minimum, maximum, mean,median, and standard deviation.

[131]:
```
# Display the numerical inforamtion in the data
display(df2_nd.describe())
```

```
          BrdIndx         Area        Round      Bright     Compact  \
count  674.000000   674.000000  674.000000  674.000000  674.000000
mean     2.022567   563.857567    1.211810  165.672745    2.160994
std      0.622502   693.909270    0.546658   62.872555    0.834991
min      1.000000    10.000000    0.000000   26.850000    1.000000
25%      1.570000   160.250000    0.822500  127.880000    1.622500
```

```
50%      1.940000   318.500000    1.180000  170.020000    1.990000
75%      2.380000   679.750000    1.500000  224.315000    2.480000
max      4.530000  5767.000000    3.520000  245.870000    8.070000

          ShpIndx      Mean_G      Mean_R    Mean_NIR        SD_G  …  \
count  674.000000  674.000000  674.000000  674.000000  674.000000  …
mean     2.266647  165.099288  162.758694  169.160178   10.561855  …
std      0.714465   60.317145   72.860031   69.236184    5.012118  …
min      1.040000   22.910000   26.520000   31.110000    3.550000  …
25%      1.710000  131.890000   99.490000  112.660000    6.920000  …
50%      2.170000  189.425000  159.525000  168.325000    8.925000  …
75%      2.680000  207.705000  237.307500  237.540000   13.105000  …
max      5.410000  246.350000  253.610000  253.630000   36.400000  …

       SD_NIR_140      LW_140   GLCM1_140    Rect_140   GLCM2_140    Dens_140  \
count  674.000000  674.000000  674.000000  674.000000  674.000000  674.000000
mean    24.362493    2.975445    0.812270    0.614659    7.984347    1.490312
std     12.341204    5.254535    0.105813    0.195224    0.767536    0.457939
min      2.650000    1.000000    0.200000    0.100000    5.690000    0.230000
25%     14.275000    1.380000    0.760000    0.460000    7.362500    1.190000
50%     22.155000    1.870000    0.830000    0.630000    7.940000    1.490000
75%     33.227500    2.627500    0.890000    0.767500    8.640000    1.850000
max     61.340000   64.700000    0.970000    1.000000    9.570000    2.410000

        Assym_140    NDVI_140  BordLngth_140    GLCM3_140
count  674.000000  674.000000     674.000000   674.000000
mean     0.644496    0.024050    1296.216617  1145.939065
std      0.249024    0.139049    1062.686078   556.731858
min      0.030000   -0.360000      34.000000   211.270000
25%      0.460000   -0.080000     542.000000   761.167500
50%      0.680000   -0.030000    1050.000000  1055.535000
75%      0.850000    0.140000    1746.000000  1411.462500
max      1.000000    0.370000    8896.000000  3806.360000

[8 rows x 147 columns]
```

1.4 Data normalization or scaling I will use the StandardScaler to scale the data and create the new columns to store the data after scaling.

[132]:
```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
standard_scaler = StandardScaler()

# Select the numerical columns to scale
numerical_columns = df2_nd.columns[1:]  # Exclude the first column assuming␣
 ↪it's non-numeric
```

```python
# Apply Standardization and create new columns
for col in numerical_columns:
    df2_nd[f'{col}_standard_scaled'] = standard_scaler.
 ↪fit_transform(df2_nd[[col]])
```

```
/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/317947378.py:11
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`
```

[134]:
```
# Check the new data set
display(df2_nd.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 674 entries, 0 to 506
Columns: 295 entries, class to GLCM3_140_standard_scaled
dtypes: float64(280), int64(14), object(1)
memory usage: 1.5+ MB
```

```
None
```

1.5 Encoding categorical data

[136]:
```
# Encode the "class" columns by using labelEncoder
from sklearn.preprocessing import LabelEncoder

# Initialize the label encoder
label_encoder = LabelEncoder()

# Encode the 'class' column
df2_nd['class_encoded'] = label_encoder.fit_transform(df2_nd['class'])

# Display the unique encoded values
print("Unique values in column 'class_encoded':")
print(df2_nd['class_encoded'].unique())
```

```
Unique values in column 'class_encoded':
[2 3 8 1 0 4 6 7 5]
```

1.6 Feature engineering Based on the analysis questions, I will create a new column named base on the valule of Normalized Difference Vegetation Index (NDVI).

[138]:
```
# Define the bins and labels for the NDVI categories
bins = [-np.inf, 0, np.inf]  # Bin edges: (-∞, 0], (0, ∞)
labels = ['Negative NDVI', 'Positive NDVI']  # Labels for the categories

# Create a new column 'NDVI_Category' based on the NDVI values
df2_nd['NDVI_Category'] = pd.cut(df2_nd['NDVI'], bins=bins, labels=labels,␣
 ↪right=True, duplicates='drop')
```

```
/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/3333994037.py:6
: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`
```

1.2.2 2. Explore the distribution of different land cover classes across the entirety of our dataset.

Pie chart and bar chart will be used the explore the distribution of the different land cover classes in the data.

[139]:
```
# Determine the frequencies of various land cover categories
land_cover_freq = df2_nd['class'].value_counts().reset_index()

# Rename the columns to 'class' and 'count'
land_cover_freq.columns = ['class', 'count']

# display(land_cover_freq)

# Select the top 5 land cover classes
top_5_land_cover = land_cover_freq.head(5)

# Create a pie chart to visualize the distribution of the top 5 land cover␣
 ↪classes
fig = px.pie(top_5_land_cover, values='count', names='class',
             title='Figure 2_1 Pie chart shows the top 5 land Ccver classes')

fig.show()
```

[140]:
```
# Create a bar chart to visualize the distribution pattern
fig = px.bar(land_cover_freq, x='class', y='count',
             title='Figure 2_2 Bar chart shows the distribution of land cover␣
 ↪classes',
             text_auto=True,
             color='class',
             labels={'Frequency': 'Frequency Count', 'Land Cover Class': 'Land␣
 ↪Cover Class'})

# Update the layout
fig.update_layout(xaxis={'categoryorder':'total descending'})  # Order␣
 ↪categories by frequency

# Show the figures
fig.show()
```

From the pie chart and bar chart above, it's evident that the most prevalent land cover class is building, accounting for 23.6% of the total frequency, followed by concrete and grass.

The descending order of land cover classes based on total counts is displayed in the bar chart. Notably, the top two land cover classes, building and concrete, are human-made, while grass and tree rank third and fourth, respectively. Conversely, car, soil, and pool represent the least prevalent land cover types, indicating significant urbanization with high vehicular and human-built infrastructure. Despite urbanization, grass and tree cover still account for a substantial portion, approximately 21.7% and 20.3%, respectively.

1.2.3 3. Investigate relationships between the mean greenness (Mean_G), mean redness (Mean_R), and mean near-infrared (Mean_NIR) values across different land cover classes

[141]:
```
# Create a 3D scatter plot
fig = px.scatter_3d(df2_nd, x='Mean_G', y='Mean_R', z='Mean_NIR', color='class',
                    title='Figure 2_3 Relationship between Mean_G, Mean_R, and␣
 ↪Mean_NIR across land vover classes',
                    labels={'Mean_G': 'Mean Greenness', 'Mean_R': 'Mean␣
 ↪Redness', 'Mean_NIR': 'Mean Near-Infrared'},
                    )

# Update the layout
fig.update_layout(scene=dict(xaxis_title='Mean Greenness', yaxis_title='Mean␣
 ↪Redness', zaxis_title='Mean Near-Infrared'))

# Update the marker size and add dark edges
fig.update_traces(marker=dict(size=5, line=dict(color='black', width=0.5)))

# Show the figure
fig.show()
```

From the 3D plot, upon closer inspection, the interaction between Mean_G (Green spectral variable), Mean_R (Red spectral variable), and Mean_NIR (Near Infrared spectral variable) across different land cover classes becomes apparent.

Pool, asphalt, shadow, and soil are distinguishable with clear clusters, facilitated by these three parameters in the scatterplot. However, building and concrete, as well as grass and tree, exhibit overlapping regions due to similar colors and materials. Additionally, most of the car records are intermingled with the building and concrete classes, although some cars exhibit distinct colors, making them stand out and identifiable using these three parameters.

1.2.4 4. Create a bubble chart to demonstrate the association between Area and Roundness, where the size of each bubble represents the Compactness of the land cover.

[142]:
```
# Map each unique class to a specific color
class_color_map = {
    'building': 'blue',
    'concrete': 'red',
    'tree': 'green',
    'asphalt': 'orange',
    'grass': 'purple',
    'shadow': 'yellow',
    'soil': 'cyan',
    'pool': 'magenta'
}
```

```
# Create hover text and bubble size
hover_text = []
bubble_size = []

# Iterate over the dataframe to create hover text and bubble size
for index, row in df2_nd.iterrows():
    hover_text.append(('Class: {class_name}<br>'+
                       'Area: {area}<br>'+
                       'Roundness: {roundness}<br>'+
                       'Compactness: {compactness}').
 ↪format(class_name=row['class'],
                                                        area=row['Area'],
                                                       ␣
 ↪roundness=row['Round'],
                                                       ␣
 ↪compactness=row['Compact']))
    bubble_size.append(row['Compact'])

# Add hover text and bubble size to the dataframe
df2_nd['text'] = hover_text
df2_nd['size'] = bubble_size

# Create the bubble chart
fig = px.scatter(df2_nd, x='Area', y='Round', color='class', size='size',␣
 ↪color_discrete_map=class_color_map, hover_data=['text'])

# Update layout
fig.update_layout(title='Figure 2_5 Bubble chart shows the association between␣
 ↪Area, Roundness, and Compactness, and color by land class',
                  xaxis_title='Area', yaxis_title='Roundness')

# Show the figure
fig.show()
```

```
/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/1514510148.py:2
9: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`

/var/folders/t0/m35j63d92_l5jr028g67rfzr0000gn/T/ipykernel_15942/1514510148.py:3
0: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling
`frame.insert` many times, which has poor performance.  Consider joining all
```

```
columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame,
use `newframe = frame.copy()`
```

From the bubble chart, it can be observed that overall, buildings have the highest area compared to other land cover classes but exhibit lower roundness, with relatively small compactness. Concrete exhibits the second highest area, with most of its roundness values falling within the middle range of 0.5 to 2. However, there are a few data points with higher roundness and smaller compactness.

Most of the other land cover classes have areas ranging between 0 to 1000 square meters, with roundness values ranging from 0 to 2.5. Some data points in the upper left corner of the chart exhibit higher roundness and larger compactness, mainly belonging to grass, trees, and concrete.

1.2.5 5. Examine the spread of average greenness (Mean_G) across various land cover categories through a box plot analysis.

[143]:
```
# Create the box plot
fig = px.box(df2_nd, x='class', y='Mean_G', color='NDVI_Category',
             title='Spread of Mean Greenness Across Land Cover Categories',
             labels={'Mean_G': 'Mean Greenness', 'class': 'Land Cover␣
 ↪Category'},
             color_discrete_map={'Negative NDVI': 'red', 'Positive NDVI':␣
 ↪'green'})

# Update the layout
fig.update_layout(xaxis_title='Land Cover Category', yaxis_title='Mean␣
 ↪Greenness')

# Show the figure
fig.show()
```

1.2.6 6.Produce a scatter plot matrix (pair plot) displaying pairwise relationships between selected features, with each scatter plot colored according to the land cover class.

[110]:
```
import plotly.figure_factory as ff

# Define the features you want to include in the scatter plot matrix
selected_features = ['Mean_G', 'Mean_R', 'Mean_NIR', 'NDVI', 'Compact']

# Concatenate the selected features and the land cover class into a single␣
 ↪DataFrame
pairplot_data = df2_nd[selected_features + ['class']]

# Create the scatter plot matrix
fig = ff.create_scatterplotmatrix(pairplot_data, diag='histogram',␣
 ↪index='class',
                                  colormap_type='cat', height=800, width=800)
```

```
# Update the layout
fig.update_layout(title='Figure 2_7 Scatter Plot Matrix of Selected Features␣
 ↪Colored by Land Cover Class')

# Show the plot
fig.show()
```

The scatter plot matrix reveals that the mean of the color and the Normalized Difference Vegetation Index (NDVI) are effective parameters for distinguishing between different land cover classes. However, there are instances where multiple classes overlap, indicating a challenge in accurately classifying certain areas.

1.2.7 7.Plot Correlation matrix of numerical features

base on the description of data set website, except features: BrdIndx: Border Index (shape variable) Area: Area in m2 (size variable) Round: Roundness (shape variable) Bright: Brightness (spectral variable) Compact: Compactness (shape variable) ShpIndx: Shape Index (shape variable) Mean_G: Green (spectral variable) Mean_R: Red (spectral variable) Mean_NIR: Near Infrared (spectral variable) SD_G: Standard deviation of Green (texture variable) SD_R: Standard deviation of Red (texture variable) SD_NIR: Standard deviation of Near Infrared (texture variable) LW: Length/Width (shape variable) GLCM1: Gray-Level Co-occurrence Matrix [i forget which type of GLCM metric this one is] (texture variable) Rect: Rectangularity (shape variable) GLCM2: Another Gray-Level Co-occurrence Matrix attribute (texture variable) Dens: Density (shape variable) Assym: Assymetry (shape variable) NDVI: Normalized Difference Vegetation Index (spectral variable) BordLngth: Border Length (shape variable) GLCM3: Another Gray-Level Co-occurrence Matrix attribute (texture variable)

These variables repeat for each coarser scale. so i will use these numeric feature to make the correlation matrix

[115]:
```
# Calculate correlation matrix
correlation_matrix = numerical_df.corr()

# Create a list of correlation values as annotations
annotations = []
for i, row in enumerate(correlation_matrix.values):
    for j, value in enumerate(row):
        annotations.append(dict(text=str(round(value, 2)),
                                x=numerical_df.columns[j],
                                y=numerical_df.columns[i],
                                xref='x1', yref='y1',
                                showarrow=False))

# Create a heatmap with annotations
fig = ff.create_annotated_heatmap(
    z=correlation_matrix.values,
    x=list(numerical_df.columns),
```

```
    y=list(numerical_df.columns),
    colorscale='Viridis',
    annotation_text=correlation_matrix.values,
    showscale=True
)

# Update layout
fig.update_layout(
    title='Figure 2_7 Correlation Matrix of Selected Numerical Features',
    xaxis_title='Features',
    yaxis_title='Features',
    annotations=annotations
)

# Show the figure
fig.show()
```

form the correaltion matrix, this is evident that the correlation between rect with brdindx, round and rect, compact and rect, glcm1 and glcm3 are high.

1.2.8 8.Create a simple dashboard using pydash.

[122]:
```
!pip install pydash
```

```
Requirement already satisfied: pydash in
/Users/xiexiaoyang/anaconda3/lib/python3.11/site-packages (7.0.7)
Requirement already satisfied: typing-extensions!=4.6.0,>=3.10 in
/Users/xiexiaoyang/anaconda3/lib/python3.11/site-packages (from pydash) (4.7.1)
```

[123]:
```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

# Initialize the Dash app
app = dash.Dash(__name__)

# Define the layout of the dashboard
app.layout = html.Div(children=[
    html.H1(children='Simple PyDash Dashboard'),

    html.Div(children='''
        Enter a number to see its square:
    '''),

    dcc.Input(id='input-number', type='number', value=5),

    html.Div(id='output-container')
```

```
])

# Define callback to update output based on input
@app.callback(
    Output('output-container', 'children'),
    [Input('input-number', 'value')]
)
def update_output(value):
    return f'The square of {value} is {value ** 2}'

# Run the app
if __name__ == '__main__':
    app.run_server(debug=True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[123], line 1
----> 1 import dash
      2 import dash_core_components as dcc
      3 import dash_html_components as html

ModuleNotFoundError: No module named 'dash'
```

[ ]: