Bivariate analysis

Bivariate analysis in data science involves the analysis of the relationship between two variables. The objective is to understand how changes in one variable are associated with changes in another variable. This type of analysis helps in exploring patterns, trends, and dependencies within the data.

Here are some common techniques used in bivariate analysis:

1.Scatter Plots
2.Correlation Analysis
3.Crosstabulation (Contingency Tables)
4.Regression Analysis
5.Box Plots
6.Heatmaps
7.Correlation Heatmaps
8.Contour Plots

Correlation Analysis:

Correlation measures the strength and direction of a linear relationship between two variables. The Pearson correlation coefficient is commonly used for continuous variables, while other correlation coefficients like Spearman or Kendall may be used for ordinal or non-parametric data.

Crosstabulation (Contingency Tables):

This technique is used when dealing with categorical variables. It shows how the frequency of occurrence of one variable changes with the change in another variable. Cross-tabulations are often displayed in a table format.

Regression Analysis:

Regression analysis helps to model and quantify the relationship between two variables. It's particularly useful when you want to predict the value of one variable based on the values of others. Simple linear regression deals with one independent variable, while multiple linear regression deals with more than one.

Box Plots:

Box plots can be used to compare the distributions of one variable across different levels of another variable. They show the median, quartiles, and potential outliers in a visually effective way.

Heatmaps:

Heatmaps are useful for visualizing the relationship between two categorical variables. They provide a color-coded representation of the relationship strength between different categories.

Correlation Heatmaps:

Correlation heatmaps are graphical representations of the correlation matrix between multiple variables. This is useful for quickly identifying strong correlations and potential multicollinearity in a dataset.

Joint Probability Distributions:

For two discrete variables, joint probability distributions provide the probability of occurrence for each combination of values. This can be represented in tabular or graphical form.

Contour Plots:

Contour plots are used for visualizing three-dimensional relationships between two continuous variables and a third dependent variable. These can provide insights into complex relationships.

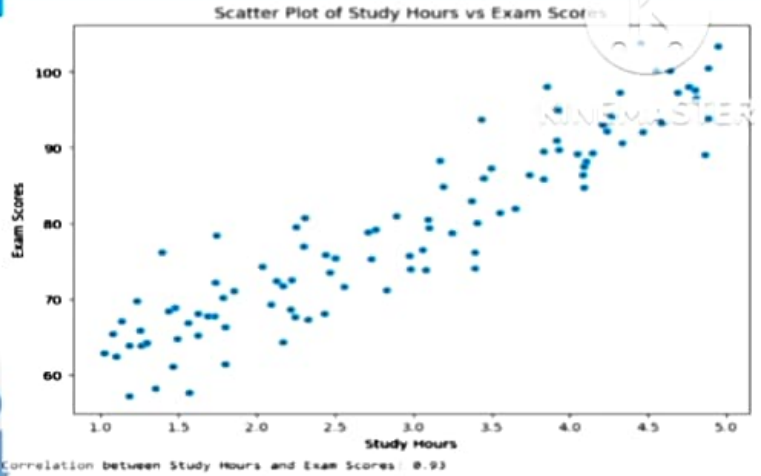
python code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Load the data
data = pd.read_csv('student_marks.csv')
# Explore the data
print(data.head())
print(data.describe())
# Bivariate analysis
# Scatter plot
plt.scatter(data['marks_in_maths'], data['marks_in_physics'])
plt.xlabel('Marks in maths')
plt.ylabel('Marks in physics')
plt.show()
# Box plot
plt.boxplot([data['marks_in_maths'], data['marks_in_physics']])
plt.xlabel('Subject')
plt.ylabel('Marks')
plt.show()
# Histogram
plt.hist(data['marks_in_maths'])
plt.xlabel('Marks in maths')
plt.ylabel('Frequency')
plt.show()
```

```
# Correlation
print(data.corr())

# Regression
# Linear regression
model = lm.LinearRegression()
model.fit(data[['marks_in_maths']], data['marks_in_physics'])

# Make predictions
y_pred = model.predict(data[['marks_in_maths']])

# Evaluate the model
print(model.score(data[['marks_in_maths']], data['marks_in_physics']))

# Plot the regression line
plt.scatter(data['marks_in_maths'], data['marks_in_physics'])
plt.plot(data[['marks_in_maths']], y_pred, color='red')
plt.xlabel('Marks in maths')
plt.ylabel('Marks in physics')
plt.show()
```

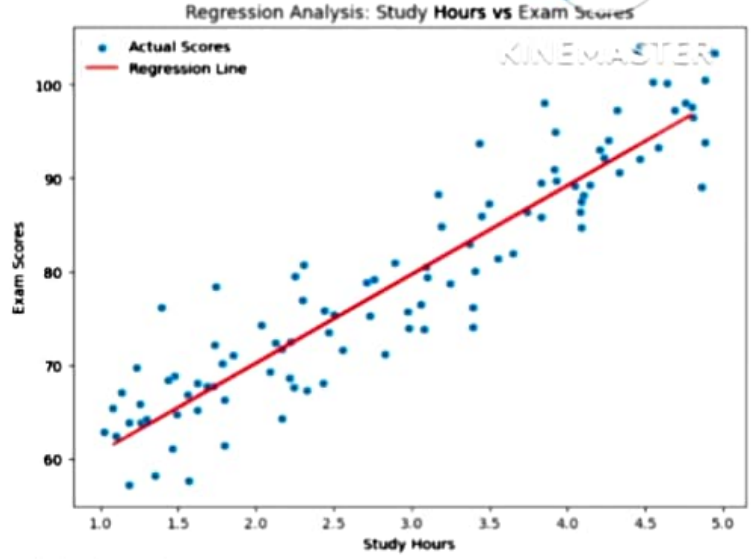
Study Hours vs Exam Scores

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics

# Generate a hypothetical student marks dataset
np.random.seed(42)
study_hours = np.random.uniform(1, 5, 100)
exam_scores = 50 + 10 * study_hours + np.random.normal(0, 5, 100)

# Create a DataFrame
data = pd.DataFrame({'Study_Hours': study_hours, 'Exam_Scores': exam_scores})

# Scatter Plot
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=data)
plt.title('Scatter Plot of Study Hours vs Exam Scores')
plt.xlabel('Study Hours')
plt.ylabel('Exam Scores')
plt.show()
```



Study Hours vs Exam Scores

```
# Correlation Analysis
correlation = data['Study_Hours'].corr(data['Exam_Scores'])
print(f'Correlation between Study Hours and Exam Scores: {correlation:.2f}')
# Regression Analysis
X = data[['Study_Hours']]
y = data['Exam_Scores']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create a linear regression model
model = LinearRegression()
# Train the model
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)
# Plot the regression line
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=data, label='Actual Scores')
plt.plot(X_test, y_pred, color='red', label='Regression Line')
plt.title('Regression Analysis: Study Hours vs Exam Scores')
plt.xlabel('Study Hours')
plt.ylabel('Exam Scores')
plt.legend()
plt.show()
# Evaluate the model
mse = metrics.mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse:.2f}')
```



Mean Squared Error: 16.34

Percentage Tables

The relationship between two variables in a table can be understood to be significant. Percentage tables(relative frequency table) provide a quick and easy way to understand the distribution of data, making it easier to identify patterns and trends. They are particularly useful for comparing relative frequencies across different categories or intervals within a dataset.

A percentage table, also known as a relative frequency table, is a tabular representation of data that expresses the relative frequencies or percentages of different categories or values within a dataset. It is often used in statistics and data analysis to provide a clear and concise summary of the distribution of a categorical variable or the distribution of values within a numerical variable.

For Categorical Variables:

Frequency Table:

Start by creating a frequency table that lists the categories or groups and their corresponding frequencies (the number of occurrences).

Percentage Calculation:

Add a column to the table to represent the percentage of each category. To calculate the percentage, divide the frequency of each category by the total number of observations and multiply by 100.

Table Format:

The table format typically includes columns for categories, frequencies, and percentages.

Percentage Table

Category	Frequency	Percentage
A	20	20%
B	30	30%
C	50	50%
Total	100	100%

Percentage Table

For Numerical Variables (Grouped Data):

Create Data Intervals (Bins):

If you are dealing with numerical data, group the data into intervals or bins. This is often done to simplify the analysis and interpretation.

Frequency Table:

Create a frequency table that shows the intervals or bins and the corresponding frequencies.

Percentage Calculation:

Add a column for percentages. Calculate the percentage for each interval by dividing its frequency by the total number of observations and multiplying by 100.

Table Format:

The table format includes columns for intervals, frequencies, and percentages.

Interval	Frequency	Percentage
0-10	15	15%
11-20	25	25%
21-30	40	40%
Total	80	100%

analysing contigency table

Analyzing a contingency table involves examining the relationship between two categorical variables. Contingency tables, also known as cross-tabulations or two-way tables, display the frequency distribution of the joint occurrences of two categorical variables.

1. Examine the Table:

Look at the rows and columns of the contingency table. Each cell represents the count or frequency of observations that fall into a specific combination of categories.

2. Row and Column Margins:

Calculate row and column totals (margins) to get an understanding of the distribution of each variable individually. This helps in identifying the most and least frequent categories.

3. Calculate Percentages:

Add percentage columns to the table. Calculate percentages for each cell by dividing the cell count by the total count in the table and multiplying by 100. This helps in understanding the relative distribution of categories.

4. Assess Independence:

Contingency tables are often used to test for the independence of the two variables. Independence means that the occurrence of one variable is not dependent on the other. Statistical tests such as the Chi-Square test can be performed to assess whether there is a significant association between the two variables.

5. Chi-Square Test:

Use the Chi-Square test to determine whether the observed distribution in the contingency table is significantly different from what would be expected if the variables were independent. The test produces a p-value, and if it is below a chosen significance level (e.g., 0.05), you may reject the null hypothesis of independence.

6. Interpretation of Results:

If the Chi-Square test indicates a significant association, further investigate the nature of the relationship. Look at specific cell frequencies and percentages to understand which categories contribute the most to the observed association.

7. Visualizations:

Create visual representations of the contingency table, such as stacked bar charts or mosaic plots, to better understand the relationship between the two categorical variables.

	Category A	Category B	Category C	Total
Group 1	15	10	5	30
Group 2	20	25	15	60
Group 3	10	15	20	45
Total	45	50	40	135

In this example, you would analyze the relationships between "Group" and "Category" variables. Calculate percentages, perform a Chi-Square test, and interpret the results to draw conclusions about the association between the two variables.

create a percentage table and a contingency table using Python

```
import pandas as pd
# Creating a hypothetical student database
data = {
    'Student_ID': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Gender': ['Male', 'Female', 'Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Male', 'Female'],
    'Grade': ['A', 'B', 'B', 'C', 'A', 'C', 'B', 'C', 'A', 'B'],
}
df = pd.DataFrame(data)
# Display the student database
print("Student Database:")
print(df)
print("\n")
# Percentage Table (Relative Frequency Table) for Gender
gender_percentage = df['Gender'].value_counts(normalize=True) * 100
print("Percentage Table for Gender:")
print(gender_percentage)
print("\n")
# Contingency Table for Gender vs Grade
contingency_table = pd.crosstab(df['Gender'], df['Grade'])
print("Contingency Table for Gender vs Grade:")
print(contingency_table)
```

```
Student Database:
   Student_ID  Gender Grade
0           1    Male     A
1           2  Female     B
2           3  Female     B
3           4    Male     C
4           5    Male     A
5           6  Female     C
6           7    Male     B
7           8  Female     C
8           9    Male     A
9          10  Female     B


Percentage Table for Gender:
Male      50.0
Female    50.0
Name: Gender, dtype: float64


Contingency Table for Gender vs Grade:
Grade   A  B  C
Gender
Female  0  3  2
Male    3  1  1
```

Handling Several Batches

Handling several batches typically refers to managing and processing data in chunks or groups, especially in the context of data analysis, machine learning, or any computational task where the dataset is large and cannot fit into memory all at once.

common approaches and considerations for handling several batches of data:

1. Batch Processing:

Data Splitting: Divide the dataset into smaller batches or chunks. The size of each batch depends on the available memory and processing capabilities.

Iterative Processing: Process one batch at a time in an iterative manner. This is common in tasks like training machine learning models, where each batch contributes to updating model parameters.

Handling Several Batches

2.Data Loading:

Streaming Data: If the data is coming in real-time or in a streaming fashion, handle each incoming batch as it arrives.

Data Loading Libraries: Utilize data loading libraries that support batch loading, such as Python's generator function, TensorFlow's tf.data API, or PyTorch's DataLoader. These tools allow you to load and process data in batches efficiently.

3. Memory Management:

Batch Size: Choose an appropriate batch size based on the available memory. A smaller batch size might be necessary if memory constraints are tight.

Data Cleanup: Release memory occupied by each batch after it has been processed to avoid memory leaks.

Handling Several Batches

4. Parallelization:

Parallel Processing: If your system allows for it, consider parallelizing the processing of different batches. This can speed up overall processing time, especially if your tasks are CPU-intensive.

Distributed Computing: In the case of very large datasets, consider distributed computing frameworks like Apache Spark for parallel and distributed processing.

5. Error Handling:

Batch-level Logging: Implement logging at the batch level to keep track of errors or anomalies. This helps in identifying issues early in the process.

Retry Mechanism: Include mechanisms to retry processing a batch in case of failures, ensuring robustness in handling unexpected errors.

Handling Several Batches

6. Monitoring and Reporting:

Progress Monitoring: Implement progress monitoring to keep track of how many batches have been processed and the overall progress of the task.

Reporting: Include reporting mechanisms to provide insights into the performance of each batch or any specific issues encountered.

7. Checkpointing:

Checkpointing: Save the state of your processing after each batch. This is particularly important for long-running tasks to resume from the last processed batch in case of interruptions.

Handling Several Batches

8. Resource Utilization:

Resource Management: Optimize the utilization of CPU, GPU, or other resources during batch processing. Ensure that resources are efficiently allocated and deallocated.

9. Testing and Validation:

Batch-level Testing: Perform testing and validation at the batch level to ensure the correctness of processing logic for each chunk of data.

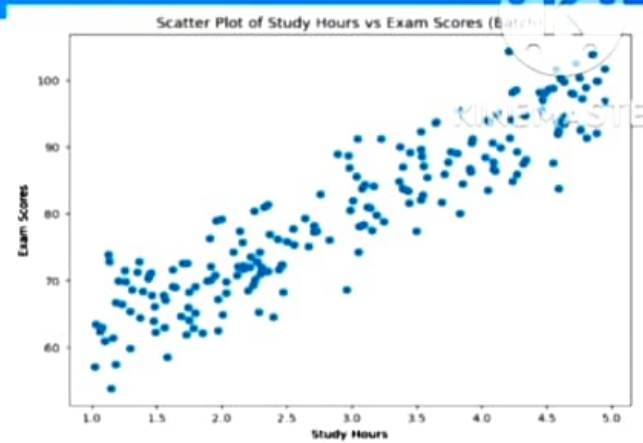
Integration Testing: Test the end-to-end workflow, considering the entire dataset processed in batches.Handling several batches efficiently is crucial for dealing with large datasets and computationally intensive tasks

Handling Several Batches

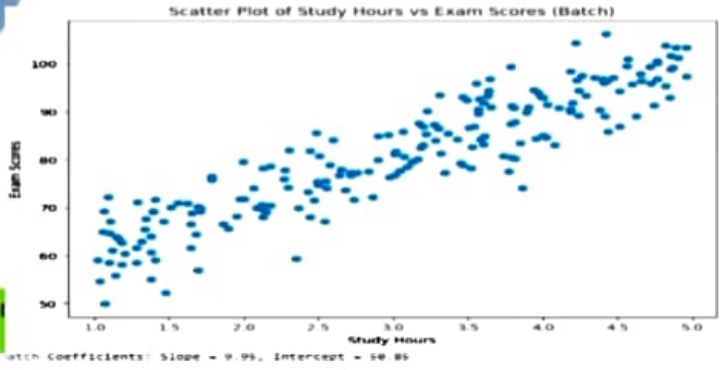
Handling several batches in the context of bivariate analysis typically involves processing data in chunks or groups, especially when dealing with large datasets. Below is an example of how you might handle several batches in Python for a bivariate analysis scenario using a hypothetical student database

Handling Several Batches

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
# Create a hypothetical student database
np.random.seed(42)
num_students = 1000
study_hours = np.random.uniform(1, 5, num_students)
exam_scores = 50 + 10 * study_hours + np.random.normal(0, 5, num_students)
student_data = pd.DataFrame({'Study_Hours': study_hours, 'Exam_Scores': exam_scores})
# Set the batch size for processing
batch_size = 200
# Process the data in batches
for batch_start in range(0, len(student_data), batch_size):
    # Select a batch of data
    batch = student_data.iloc[batch_start:batch_start + batch_size]
    # Bivariate Analysis for the current batch
    # Scatter Plot
    plt.figure(figsize=(8, 6))
    plt.scatter(batch['Study_Hours'], batch['Exam_Scores'])
    plt.title('Scatter Plot of Study Hours vs Exam Scores (Batch)')
    plt.xlabel('Study Hours')
    plt.ylabel('Exam Scores')
    plt.show()
```



Batch Coefficients: Slope = 9.34, Intercept = 52.63

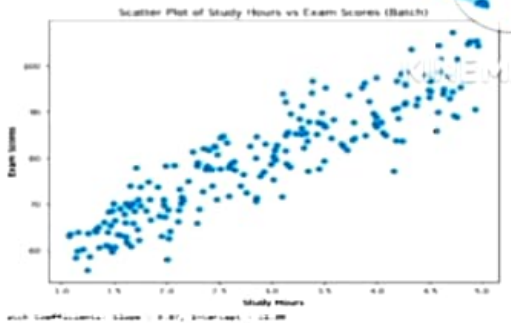


Batch Coefficients: Slope = 9.95, Intercept = 50.85

Virtual classroo

Handling Several Batches

```
# Linear Regression
X = batch[['Study_Hours']]
y = batch['Exam_Scores']

model = LinearRegression()
model.fit(X, y)

# Print coefficients
print(f'Batch Coefficients: Slope = {model.coef_[0]:.2f}, Intercept = {model.intercept_:.2f}')
```




```
# Continue with the analysis or reporting for the entire dataset
# ...
# Visualize the overall regression line
plt.figure(figsize=(8, 6))
plt.scatter(student_data['Study_Hours'], student_data['Exam_Scores'], alpha=0.5, label='All Data')
plt.plot(student_data['Study_Hours'], model.predict(student_data[['Study_Hours']]), color='red', label='Overall Regression Line')
plt.title('Overall Regression Line for Study Hours vs Exam Scores')
plt.xlabel('Study Hours')
plt.ylabel('Exam Scores')
plt.legend()
plt.show()
```

Handling Several Batches

```
# Linear Regression
X = batch[['Study_Hours']]
y = batch['Exam_Scores']

model = LinearRegression()
model.fit(X, y)

# Print coefficients
print(f'Batch Coefficients: Slope = {model.coef_[0]:.2f}, Intercept = {model.intercept_:.2f}')
```


```
# Continue with the analysis or reporting for the entire dataset
# ...
# Visualize the overall regression line
plt.figure(figsize=(8, 6))
plt.scatter(student_data['Study_Hours'], student_data['Exam_Scores'], alpha=0.5, label='All Data')
plt.plot(student_data['Study_Hours'], model.predict(student_data[['Study_Hours']]), color='red', label='Overall Regression Line')
plt.title('Overall Regression Line for Study Hours vs Exam Scores')
plt.xlabel('Study Hours')
plt.ylabel('Exam Scores')
plt.legend()
plt.show()
```

Handling Several Batches

This approach allows you to analyze large datasets in smaller, manageable chunks, making it possible to handle situations where the entire dataset might not fit into memory at once.Handling several batches in the context of bivariate analysis typically involves processing data in chunks or groups, especially when dealing with large datasets.

create a hypothetical student database with study hours and exam scores.

We set a batch_size to define the number of data points processed in each batch.

We iterate through the data in batches, performing bivariate analysis (scatter plot and linear regression) on each batch.

The coefficients of the linear regression model are printed for each batch.

After processing all batches, we visualize the overall regression line for the entire dataset.

scatter plots and Resistant Lines

In bivariate analysis, scatter plots are often used to visualize the relationship between two variables. A resistant line (also known as robust regression line) is a line that is less sensitive to outliers compared to the ordinary least squares (OLS) regression line. Robust regression methods are designed to provide more accurate estimates when the data contains outliers or influential points.

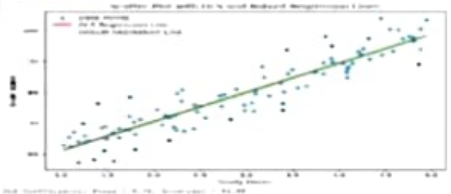
I'll use a dataset with study hours and exam scores, similar to the previous examples.

scatter plots and Resistant Lines

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm

# Create a hypothetical student database
np.random.seed(42)
num_students = 100
study_hours = np.random.uniform(1, 5, num_students)
exam_scores = 50 + 10 * study_hours + np.random.normal(0, 5, num_students)

student_data = pd.DataFrame({'Study_Hours': study_hours, 'Exam_Scores': exam_scores})

# Create a scatter plot
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=student_data, label='Data Points')

# Fit an Ordinary Least Squares (OLS) regression line
ols_model = sm.OLS(student_data['Exam_Scores'], sm.add_constant(student_data['Study_Hours'])).fit()
ols_line = np.polyval([ols_model.params['Study_Hours'], ols_model.params['const']], student_data['Study_Hours'])
plt.plot(student_data['Study_Hours'], ols_line, color='red', label='OLS Regression Line')
```

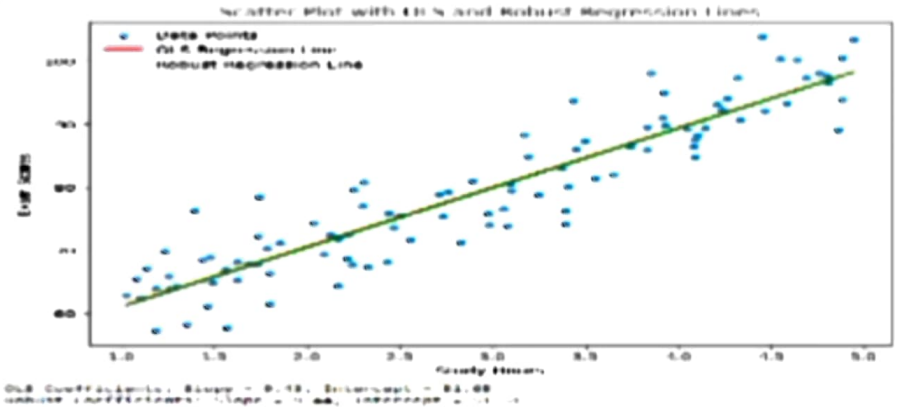
scatter plots and Resistant Lines

```
# Fit a Robust Regression line (resistant to outliers)
robust_model = sm.RLM(student_data['Exam_Scores'], sm.add_constant(student_data['Study_Hours'])).fit()
robust_line = np.polyval([robust_model.params['Study_Hours'], robust_model.params['const']], student_data['Study_Hours'])
plt.plot(student_data['Study_Hours'], robust_line, color='green', label='Robust Regression Line')

plt.title('Scatter Plot with OLS and Robust Regression Lines')
plt.xlabel('Study Hours')
plt.ylabel('Exam Scores')
plt.legend()
plt.show()

# Print coefficients
print(f'OLS Coefficients: Slope = {ols_model.params["Study_Hours"]:.2f}, Intercept = {ols_model.params["const"]:.2f}')
print(f'Robust Coefficients: Slope = {robust_model.params["Study_Hours"]:.2f}, Intercept = {robust_model.params["const"]:.2f}')
```



rams["Study_Hours"]:.2f}, Intercept = {ols_model.params["const"]:.2f}')

odel.params["Study_Hours"]:.2f}, Intercept = {robust_model.params["const"]:.2f}')



Virtual classroom Lectures

scatter plots and Resistant Lines

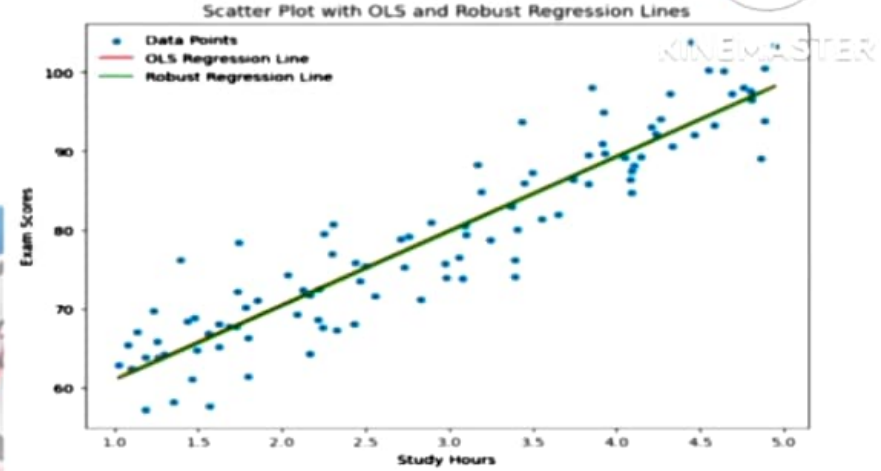
create a hypothetical student database with study hours and exam scores.

We use Seaborn to create a scatter plot of the data.

We fit both an Ordinary Least Squares (OLS) regression line and a Robust Regression line to the data using the Statsmodels library.

We plot both regression lines on the scatter plot.

We print the coefficients of both regression lines.
The Robust Regression line is less influenced by outliers, making it a more suitable choice when dealing with datasets that may contain influential points.



Transformations in bivariate analysis

Transformations in bivariate analysis involve modifying one or both variables to better meet the assumptions of statistical tests or to reveal patterns in the data. Here are some common types of transformations used in bivariate analysis, along with examples in Python using a hypothetical dataset with study hours and exam scores:

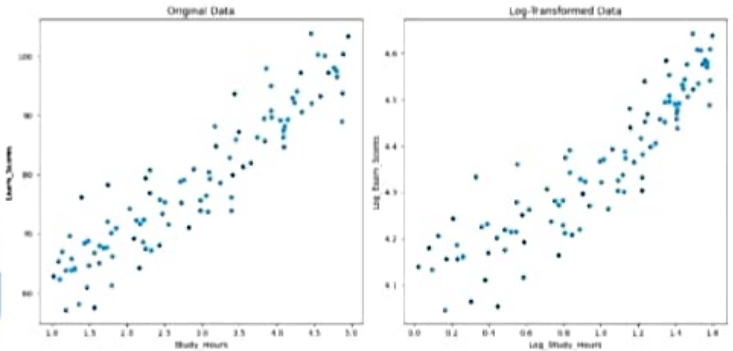
Log Transformation:

Used when the data exhibits exponential growth or decay.

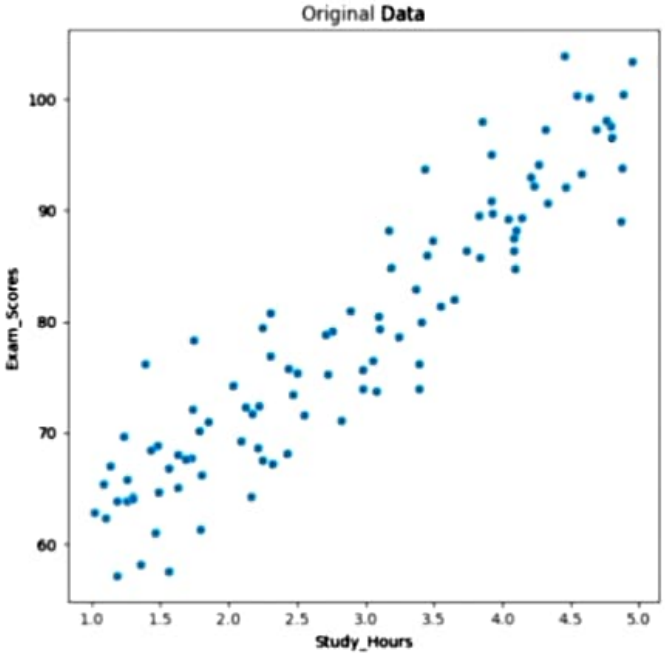
It can stabilize variance and make the relationship more linear.

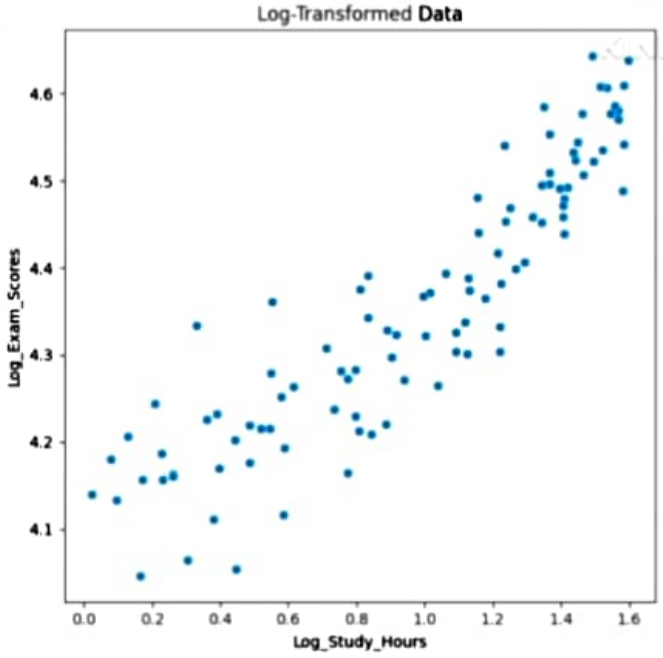
Log transformation

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
# Create a hypothetical student database
np.random.seed(42)
num_students = 100
study_hours = np.random.uniform(1, 5, num_students)
exam_scores = 50 + 10 * study_hours + np.random.normal(0, 5, num_students)
student_data = pd.DataFrame({'Study_Hours': study_hours, 'Exam_Scores': exam_scores})
# Log transformation
student_data['Log_Study_Hours'] = np.log(student_data['Study_Hours'])
student_data['Log_Exam_Scores'] = np.log(student_data['Exam_Scores'])
# Create a scatter plot with log-transformed variables
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=student_data)
plt.title('Original Data')

plt.subplot(1, 2, 2)
sns.scatterplot(x='Log_Study_Hours', y='Log_Exam_Scores', data=student_data)
plt.title('Log-Transformed Data')

plt.tight_layout()
plt.show()
```



log transformation



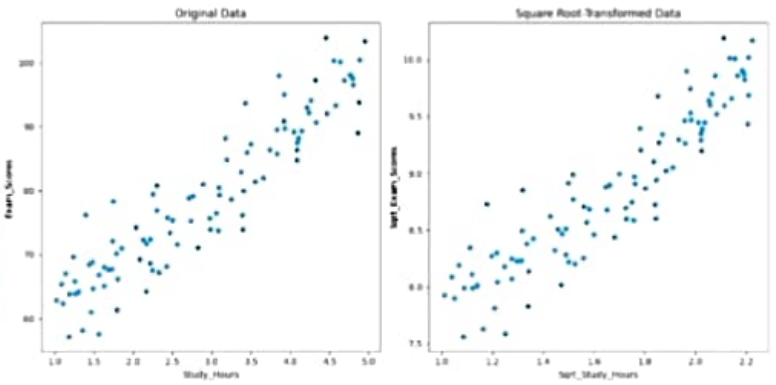


Square Root Transformation

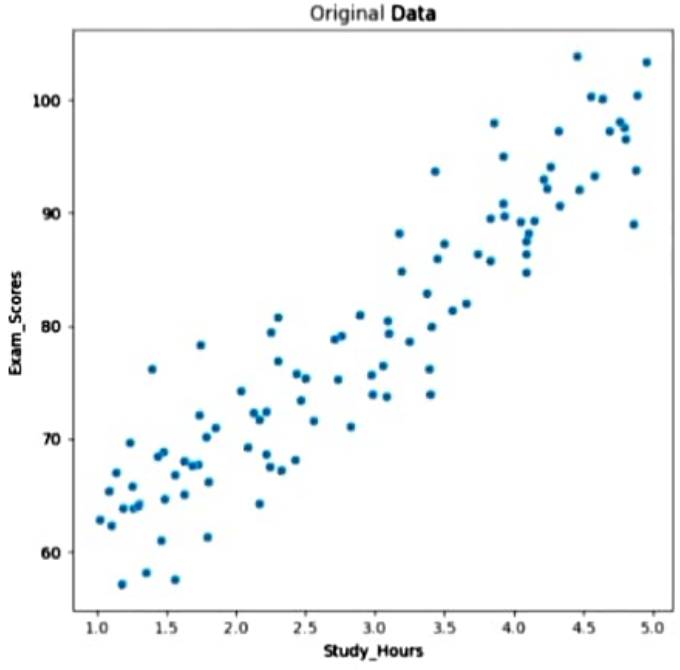
Useful when the data exhibits quadratic relationships.

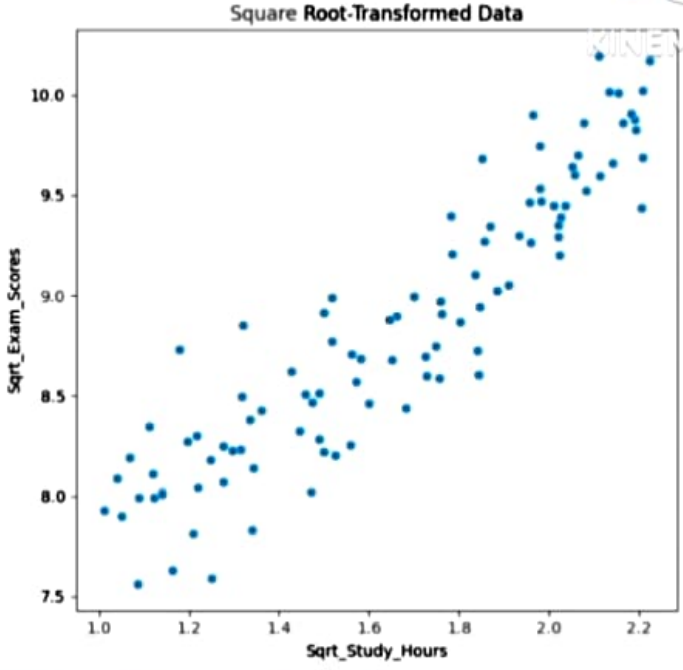
It can stabilize variance and make the relationship more linear.

```
# Square root transformation
student_data['Sqrt_Study_Hours'] = np.sqrt(student_data['Study_Hours'])
student_data['Sqrt_Exam_Scores'] = np.sqrt(student_data['Exam_Scores'])

# Create a scatter plot with square root-transformed variables
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=student_data)
plt.title('Original Data')

plt.subplot(1, 2, 2)
sns.scatterplot(x='Sqrt_Study_Hours', y='Sqrt_Exam_Scores', data=student_data)
plt.title('Square Root-Transformed Data')

plt.tight_layout()
plt.show()
```



Square Root Transformation



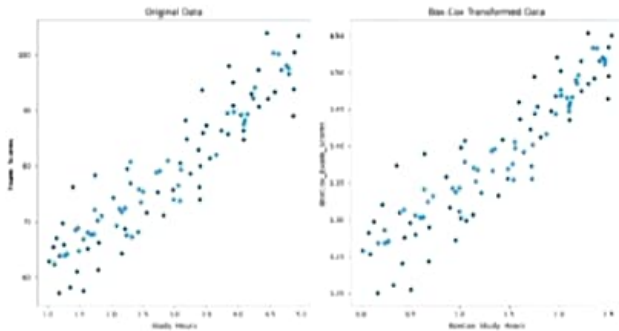


Box-Cox Transformation

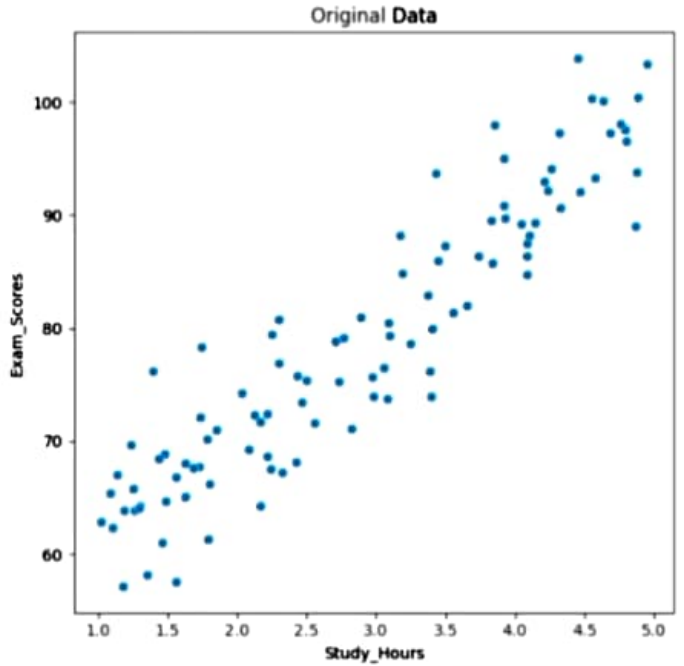
Used when dealing with non-constant variance.

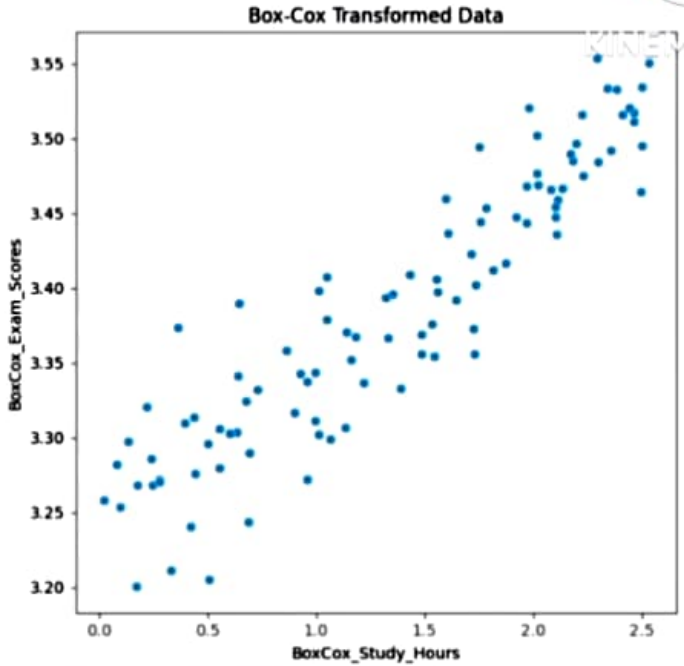
It can handle both positive and negative values.

```
from scipy.stats import boxcox

# Box-Cox transformation
student_data['BoxCox_Study_Hours'], lambda_study_hours = boxcox(student_data['Study_Hours'])
student_data['BoxCox_Exam_Scores'], lambda_exam_scores = boxcox(student_data['Exam_Scores'])

# Create a scatter plot with Box-Cox transformed variables
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.scatterplot(x='Study_Hours', y='Exam_Scores', data=student_data)
plt.title('Original Data')

plt.subplot(1, 2, 2)
sns.scatterplot(x='BoxCox_Study_Hours', y='BoxCox_Exam_Scores', data=student_data)
plt.title('Box-Cox Transformed Data')

plt.tight_layout()
plt.show()
```



Box-Cox Transformation





transformations summary

These transformations can be beneficial when the assumptions of linear regression or correlation analysis are not met, and they can help uncover more meaningful relationships in the data. The choice of transformation depends on the characteristics of your data and the specific objectives of your analysis.

Bivariate analysis involves the examination and analysis of the relationship between two variables. This type of analysis is fundamental in statistics and data science as it helps to uncover patterns, associations, and dependencies between two quantitative or categorical variables. Understanding the relationship between variables is crucial for making informed decisions, predicting outcomes, and gaining insights into the underlying structure of the data.

Types of Bivariate Data

Types of Bivariate Data:

Quantitative-Quantitative (Numeric-Numeric): Analyzing the relationship between two numerical variables. For example, studying the correlation between study hours and exam scores.

Categorical-Categorical (Category-Category): Examining the association between two categorical variables. For instance, analyzing the relationship between gender and academic performance categories.

Quantitative-Categorical (Numeric-Category): Investigating how a numerical variable varies across different categories. For example, studying how exam scores vary across different study hour categories.

Techniques in Bivariate Data

2. Techniques in Bivariate Analysis:

Scatter Plots:

A graphical representation of the relationship between two quantitative variables. Each point on the plot represents an observation in the dataset.

Correlation Analysis:

Measures the strength and direction of a linear relationship between two numerical variables. The Pearson correlation coefficient is commonly used for this purpose.

Regression Analysis:

Models and quantifies the relationship between two variables, typically involving predicting one variable based on the other. Simple linear regression deals with one independent variable, while multiple linear regression involves more than one.

Techniques in Bivariate Data

Crosstabulation (Contingency Tables):

Summarizes the relationship between two categorical variables by showing the frequency distribution of their joint occurrences.

Chi-Square Test:

Used to determine if there is a significant association between two categorical variables in a contingency table.

Transformations:

Applying mathematical transformations to variables (e.g., log, square root) to address issues like non-linearity or non-constant variance.

Techniques in Bivariate Data

Resistant Lines:

Robust regression lines that are less sensitive to outliers compared to ordinary least squares (OLS) regression lines.

Analysis of Variance (ANOVA):

Used to assess the impact of a categorical variable on a quantitative variable.

Techniques in Bivariate Data

Crosstabulation (Contingency Tables):

Summarizes the relationship between two categorical variables by showing the frequency distribution of their joint occurrences.

Chi-Square Test:

Used to determine if there is a significant association between two categorical variables in a contingency table.

Transformations:

Applying mathematical transformations to variables (e.g., log, square root) to address issues like non-linearity or non-constant variance.

Goals of Bivariate Analysis

Goals of Bivariate Analysis:

1.Understand Relationships:

Investigate how changes in one variable relate to changes in another.

2.Predictive Modeling:

Build models to predict one variable based on the values of another.

3.Identify Patterns:

Uncover patterns and trends within the data.

4.Assess Dependencies:

Determine the extent to which two variables depend on each other.