**SECTION 1: Mike (35)**

* Data Cleaning
* Exploration
* Visualization

**SECTION 2: Ridewaan (35)**

* Visualization
* Statistical Analysis
* Machine Learning

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

**MY NOTES**

* Collection and Preparation
* Exploration
* Visualization
* Applied Statistical Analysis
* Machine Learning

# DATA CLEANING & PREPERATION:

# Import necessary libraries for data manipulation and analysis

*import pandas as pd*

*import numpy as np*

# Load your dataset into a Pandas DataFrame from a CSV file

*df = pd.read\_csv('data.csv')*

# Inspect the first few rows of the DataFrame to get an overview of the data

*df.head()*

# Get information about the DataFrame, including data types and missing values

*df.info()*

# Generate basic statistics for numerical columns in the DataFrame

*df.describe()*

# Check for missing values in DataFrame and calculate the count of missing values in each column

*df.isna().sum()*

# Remove rows with missing values (NaN) from the DataFrame

*df.dropna()*

# Fill missing values in the DataFrame with a specific value (e.g., fill with the mean)

*df.fillna(value)*

# Convert the data type of a specific column to a new data type

*df['column\_name'] = df['column\_name'].astype('new\_dtype')*

# Rename a column in the DataFrame

*df.rename(columns={'old\_name': 'new\_name'}, inplace=True)*

# Identify and count duplicate rows in the DataFrame

*df.duplicated().sum()*

# Remove duplicate rows from the DataFrame

*df.drop\_duplicates(inplace=True)*

# Create a new feature by performing operations on existing columns

*df['new\_feature'] = df['feature1'] + df['feature2']*

# Scale or normalize numerical features to have similar scales (e.g., using Min-Max scaling)

*from sklearn.preprocessing import MinMaxScaler*

*scaler = MinMaxScaler()*

*df['scaled\_feature'] = scaler.fit\_transform(df[['feature']])*

# Encode categorical variables into numerical form using one-hot encoding

*df = pd.get\_dummies(df, columns=['categorical\_column'], prefix=['prefix'])*

# Split the dataset into training and testing sets for machine learning

*from sklearn.model\_selection import train\_test\_split*

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)*

# Create data visualizations to explore and understand the data

*import matplotlib.pyplot as plt*

*import seaborn as sns*

*sns.histplot(df['numeric\_feature'])*

*plt.show()*

# DATA EXPLORATION:

# Import necessary libraries for data exploration and visualization

*import pandas as pd*

*import numpy as np*

*import matplotlib.pyplot as plt*

*import seaborn as sns*

# Load your dataset into a Pandas DataFrame from a CSV file

*df = pd.read\_csv('data.csv')*

# Get the number of rows and columns in the DataFrame

*num\_rows, num\_cols = df.shape*

# Get the names of all columns in the DataFrame

*column\_names = df.columns*

# Get the data types of each column in the DataFrame

*data\_types = df.dtypes*

# Get the unique values in a specific column

*unique\_values = df['column\_name'].unique()*

# Calculate summary statistics for a specific column

*summary\_stats = df['numeric\_column'].describe()*

# Calculate the correlation matrix between numerical columns

*correlation\_matrix = df.corr()*

# Create a heatmap to visualize the correlation matrix

*plt.figure(figsize=(10, 8))*

*sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')*

*plt.title('Correlation Heatmap')*

*plt.show()*

# Count the number of occurrences of unique values in a column

*value\_counts = df['categorical\_column'].value\_counts()*

# Create a bar plot to visualize the distribution of categorical values

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

*sns.countplot(data=df, x='categorical\_column')*

*plt.title('Categorical Value Counts')*

*plt.xticks(rotation=45)*

*plt.show()*

# Group the data by a categorical column and calculate summary statistics for each group

*grouped\_data = df.groupby('grouping\_column')['numeric\_column'].mean()*

# Create a bar plot to visualize the mean values for each group

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

*sns.barplot(x=grouped\_data.index, y=grouped\_data.values)*

*plt.title('Mean Value by Group')*

*plt.xticks(rotation=45)*

*plt.show()*

# Create a scatter plot to explore the relationship between two numerical columns

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

*sns.scatterplot(data=df, x='numeric\_column1', y='numeric\_column2')*

*plt.title('Scatter Plot')*

*plt.show()*

# Create a box plot to visualize the distribution of a numerical column

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

*sns.boxplot(data=df, y='numeric\_column')*

*plt.title('Box Plot')*

*plt.show()*

# Create a pair plot to visualize pairwise relationships between numerical columns

*sns.pairplot(df[['numeric\_column1', 'numeric\_column2', 'numeric\_column3']])*

*plt.title('Pair Plot')*

*plt.show()*

# DATA VISUALIZATION:

## PLOTS:

# Import necessary libraries for data visualization

*import matplotlib.pyplot as plt*

*import seaborn as sns*

# Create a simple line plot

*plt.plot(x\_data, y\_data)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Line Plot Title')*

*plt.show()*

# Create a scatter plot

*plt.scatter(x\_data, y\_data, c='color', marker='marker\_style', s=size, label='label')*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Scatter Plot Title')*

*plt.legend()*

*plt.show()*

# Create a bar plot

*plt.bar(x\_data, y\_data, color='color', width=bar\_width)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Bar Plot Title')*

*plt.xticks(rotation=45)*

*plt.show()*

# Create a horizontal bar plot

*plt.barh(y\_data, x\_data, color='color', height=bar\_height)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Horizontal Bar Plot Title')*

*plt.show()*

# Create a histogram

*plt.hist(data, bins=number\_of\_bins, color='color', alpha=transparency)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Histogram Title')*

*plt.show()*

# Create a box plot

*plt.boxplot(data, vert=vertical, notch=True, sym='marker', labels=box\_labels)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Box Plot Title')*

*plt.show()*

# Create a violin plot

*sns.violinplot(x='x\_data', y='y\_data', data=df, hue='hue', split=True)*

*plt.xlabel('X-Axis Label')*

*plt.ylabel('Y-Axis Label')*

*plt.title('Violin Plot Title')*

*plt.show()*

# Create a pair plot (scatter matrix) to visualize pairwise relationships between numerical columns

*sns.pairplot(df[['numeric\_column1', 'numeric\_column2', 'numeric\_column3']], hue='hue', markers='markers')*

*plt.title('Pair Plot Title')*

*plt.show()*

# Create a heatmap to visualize a correlation matrix

*sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')*

*plt.title('Heatmap Title')*

*plt.show()*

## CHANGING THINGS ON PLOTS:

# Customize axis limits and ticks

*plt.xlim(x\_min, x\_max)*

*plt.ylim(y\_min, y\_max)*

*plt.xticks(x\_ticks, x\_labels)*

*plt.yticks(y\_ticks, y\_labels)*

# Add text or annotations to the plot

*plt.text(x, y, 'Text', fontsize=font\_size, color='color')*

*plt.annotate('Annotation', xy=(x, y), xytext=(x\_text, y\_text), arrowprops=dict(arrowstyle='->'))*

# Customize plot aesthetics

*plt.figure(figsize=(width, height))*

*plt.grid(True)*

*plt.legend(loc='location')*

*plt.tight\_layout()*

# Save the plot to a file

*plt.savefig('output\_plot.png', dpi=dpi)*

# Show the plot

*plt.show()*

## MORE ADVANCED PLOTS:

# Create a simple line plot

*plt.plot(x\_data, y\_data, linestyle='-', marker='o', markersize=8, color='blue', label='Line Label')*

*plt.xlabel('X-Axis Label', fontsize=12, color='black')*

*plt.ylabel('Y-Axis Label', fontsize=12, color='black')*

*plt.title('Line Plot Title', fontsize=14, color='black')*

*plt.legend(fontsize=10, loc='best')*

*plt.grid(True, linestyle='--', alpha=0.5)*

*plt.xticks(x\_ticks, x\_labels, rotation=45, fontsize=10, color='black')*

*plt.yticks(fontsize=10, color='black')*

*plt.axvline(x=0, color='red', linestyle='--', linewidth=2)*

*plt.axhline(y=0, color='green', linestyle='-.', linewidth=2)*

*plt.annotate('Important Point', xy=(x\_point, y\_point), xytext=(text\_x, text\_y), fontsize=10,*

*arrowprops=dict(arrowstyle='->', linewidth=1, color='black'), color='black')*

*plt.show()*

# Create a scatter plot

*plt.scatter(x\_data, y\_data, c='color', marker='marker\_style', s=size, label='label', alpha=0.8, edgecolors='k')*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.title('Scatter Plot Title', fontsize=14)*

*plt.legend(fontsize=10, loc='best')*

*plt.grid(True, linestyle='--', alpha=0.5)*

*plt.xticks(x\_ticks, x\_labels, rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a bar plot

*plt.bar(x\_data, y\_data, color='color', width=bar\_width, edgecolor='black', linewidth=1, label='label')*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.title('Bar Plot Title', fontsize=14)*

*plt.legend(fontsize=10, loc='best')*

*plt.grid(axis='y', linestyle='--', alpha=0.5)*

*plt.xticks(rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a horizontal bar plot

*plt.barh(y\_data, x\_data, color='color', height=bar\_height, edgecolor='black', linewidth=1, label='label')*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.title('Horizontal Bar Plot Title', fontsize=14)*

*plt.legend(fontsize=10, loc='best')*

*plt.grid(axis='x', linestyle='--', alpha=0.5)*

*plt.xticks(rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a histogram

*plt.hist(data, bins=number\_of\_bins, color='color', alpha=transparency, edgecolor='black', linewidth=1)*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.title('Histogram Title', fontsize=14)*

*plt.grid(axis='y', linestyle='--', alpha=0.5)*

*plt.xticks(rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a box plot

*plt.boxplot(data, vert=vertical, notch=True, sym='marker', labels=box\_labels)*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.title('Box Plot Title', fontsize=14)*

*plt.xticks(rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a violin plot

*sns.violinplot(x='x\_data', y='y\_data', data=df, hue='hue', split=True, inner='stick')*

*plt.xlabel('X-Axis Label', fontsize=12)*

*plt.ylabel('Y-Axis Label', fontsize=12)*

*plt.title('Violin Plot Title', fontsize=14)*

*plt.legend(fontsize=10, loc='best')*

*plt.xticks(rotation=45, fontsize=10)*

*plt.yticks(fontsize=10)*

*plt.show()*

# Create a pair plot (scatter matrix) to visualize pairwise relationships between numerical columns

*sns.pairplot(df[['numeric\_column1', 'numeric\_column2', 'numeric\_column3']], hue='hue', markers='markers')*

*plt.title('Pair Plot Title', fontsize=14)*

*plt.show()*

# Create a heatmap to visualize a correlation matrix

*sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')*

*plt.title('Heatmap Title', fontsize=14)*

*plt.show()*

# APPLIED STATISTICAL ANALYSIS:

# Calculate the mean (average) of the dataset

# The mean represents the central tendency of a dataset. A high mean indicates larger values, while a low mean indicates smaller values. For example, a mean of 10 for test scores could be considered high, while a mean of 3 might be considered low.

**Example**: A mean of 8.5 for exam scores in a class of 10 students.

*mean = np.mean(data)*

# Calculate the median of the dataset

# The median is the middle value when the data is sorted. A high median suggests that the central data points are higher, while a low median suggests lower central data points. For example, a median income of $50,000 might be considered high, while $30,000 could be considered low.

**Example**: A median household income of $45,000 in a neighborhood.

median = np.median(data)

# Calculate the mode of the dataset

# The mode represents the most frequently occurring value(s) in the dataset. A high mode frequency indicates a strong central tendency for that value. For example, a mode of 5 for car colors in a parking lot might indicate a popular color, while a mode of 1 suggests a less common color.

**Example**: The mode of car colors in a parking lot is 'blue' with a frequency of 20.

mode = stats.mode(data)

# Calculate the variance of the dataset

# Variance measures the spread or dispersion of data points around the mean. A high variance indicates greater variability, while a low variance suggests less variability. For example, a high variance in test scores may indicate that scores are spread out, while a low variance suggests scores are close together.

**Example**: Variance of 25 for test scores in a class with a mean of 50.

*variance = np.var(data)*

# Calculate the standard deviation of the dataset

# The standard deviation is the square root of the variance. It measures the average distance of data points from the mean. A high standard deviation indicates a wide spread, while a low standard deviation indicates a narrow spread. For example, a high standard deviation for employee salaries suggests a wide range of salaries, while a low standard deviation implies similar salaries.

**Example**: Standard deviation of $10,000 for employee salaries.

*std\_deviation = np.std(data)*

# Calculate the correlation coefficient between two datasets

# The correlation coefficient measures the strength and direction of the linear relationship between two variables. A high positive value (close to 1) indicates a strong positive correlation, while a high negative value (close to -1) indicates a strong negative correlation. A value close to 0 suggests a weak or no linear correlation. For example, a correlation coefficient of 0.8 between advertising spending and sales suggests a strong positive relationship, while -0.5 suggests a moderate negative relationship.

**Example**: Correlation coefficient of 0.85 between hours spent studying and exam scores.

*correlation\_coefficient = np.corrcoef(data1, data2)*

# Perform a t-test to compare means of two datasets

# A high t-statistic and a low p-value suggest that the means of two datasets are significantly different, indicating a strong difference between the groups. A low t-statistic and a high p-value suggest no significant difference. For example, in a drug trial, a low p-value (e.g., p < 0.05) indicates that the drug has a significant effect compared to a placebo.

**Example**: t-statistic of 2.5 and p-value of 0.02 in a drug trial.

*t\_statistic, p\_value = stats.ttest\_ind(data1, data2)*

# Perform linear regression analysis

# Linear regression aims to model the relationship between two variables with a linear equation. A high R-squared value (close to 1) indicates a strong linear relationship, while a low R-squared value suggests a weak relationship. For example, an R-squared value of 0.8 suggests that 80% of the variation in sales can be explained by advertising spending, indicating a strong relationship.

**Example**: An R-squared value of 0.85 in a linear regression model predicting house prices based on square footage.

*X = sm.add\_constant(data1) # Add a constant for the intercept*

*model = sm.OLS(data2, X).fit() # Fit a linear regression model*

*model\_summary = model.summary() # Get a summary of the regression results*

# Perform chi-square test for independence

# The chi-square test determines if there is a significant association between two categorical variables. A high chi-square statistic and a low p-value indicate dependence, while a low statistic and a high p-value suggest independence. For example, in a survey, a low p-value (e.g., p < 0.05) suggests that gender and preference for a particular product are dependent.

**Example**: Chi-square statistic of 20.3 and p-value of 0.001 in a contingency table analysis.

*chi2\_statistic, p\_value, dof, expected = stats.chi2\_contingency(contingency\_table)*

# Perform ANOVA (Analysis of Variance)

# ANOVA tests if there are significant differences in means between three or more groups. A high F-statistic and a low p-value suggest at least one group differs significantly from the rest. A low F-statistic and a high p-value suggest no significant differences. For example, in a study comparing the effects of different diets on weight loss, a low p-value indicates that at least one diet is significantly different from the others.

**Example**: F-statistic of 5.2 and p-value of 0.002 in an ANOVA comparing the effects of three different diets on weight loss.

*F\_statistic, p\_value = stats.f\_oneway(data1, data2, data3)*

# Perform Mann-Whitney U test for non-parametric data

# The Mann-Whitney U test assesses if two independent groups have different distributions. A low p-value indicates significant differences between the groups. For example, in a study comparing the effectiveness of two different treatments, a low p-value suggests that one treatment is significantly better than the other.

**Example**: U-statistic of 120 and p-value of 0.01 in a Mann-Whitney U test comparing treatment A and treatment B.

*U\_statistic, p\_value = stats.mannwhitneyu(data1, data2)*

# Perform Kruskal-Wallis H test for non-parametric data

# Kruskal-Wallis tests if there are significant differences in medians among three or more groups. A low H-statistic and a high p-value suggest no significant differences, while the opposite suggests significant differences. For example, in a study comparing the pain relief provided by three different pain medications, a low p-value indicates that at least one medication is significantly different from the others.

**Example**: H-statistic of 8.4 and p-value of 0.02 in a Kruskal-Wallis test comparing three different pain medications.

*H\_statistic, p\_value = stats.kruskal(data1, data2, data3)*

# Perform chi-square goodness-of-fit test

# The chi-square goodness-of-fit test checks if observed categorical data fits an expected distribution. A low chi-square statistic and a high p-value indicate a good fit, while the opposite suggests a poor fit. For example, in a genetics study, a high p-value suggests that observed genotype frequencies match expected frequencies, indicating a good fit.

**Example**: Chi-square statistic of 6.1 and p-value of 0.11 in a goodness-of-fit test for genotype frequencies.

*chi2\_statistic, p\_value = stats.chisquare(observed\_frequencies, expected\_frequencies)*

# Perform one-sample t-test

# The one-sample t-test determines if a sample mean is significantly different from a known population mean. A high t-statistic and a low p-value suggest a significant difference, while the opposite suggests no significant difference. For example, in a manufacturing process, a low p-value suggests that the product's weight differs significantly from the target weight.

**Example**: t-statistic of 3.2 and p-value of 0.003 in a one-sample t-test comparing sample mean weight to a target weight.

*t\_statistic, p\_value = stats.ttest\_1samp(data, pop\_mean)*

# Perform paired t-test

# A paired t-test compares means of two related groups (e.g., before and after measurements). A high t-statistic and a low p-value indicate a significant difference, while the opposite suggests no significant difference. For example, in a study measuring the effect of a training program on fitness, a low p-value suggests a significant improvement in fitness.

**Example**: t-statistic of 4.8 and p-value of 0.001 in a paired t-test comparing pre-training and post-training fitness levels.

*t\_statistic, p\_value = stats.ttest\_rel(before\_data, after\_data)*

# Perform Pearson correlation test

# The Pearson correlation test assesses the strength and direction of a linear relationship between two continuous variables. A high correlation coefficient and a low p-value suggest a strong linear relationship. For example, in a marketing campaign analysis, a high correlation between ad spend and sales revenue suggests that increased spending leads to higher sales.

**Example**: Correlation coefficient of 0.85 and p-value of 0.001 indicating a strong positive correlation between advertising spend and sales revenue.

*correlation\_coefficient, p\_value = stats.pearsonr(data1, data2)*

# Perform Spearman rank correlation test

# The Spearman rank correlation test assesses the strength and direction of a monotonic relationship between two variables. A high correlation coefficient and a low p-value suggest a strong monotonic relationship. For example, in a study analyzing the relationship between age and ranking in a sports competition, a high correlation suggests that older participants tend to rank higher.

**Example**: Correlation coefficient of 0.75 and p-value of 0.002 indicating a strong monotonic relationship between participant age and ranking.

*correlation\_coefficient, p\_value = stats.spearmanr(data1, data2)*

## EXTRA EXPLANATION:

# Calculate the mean (average) of the dataset

# Parameter: The dataset itself. Common Range: The mean typically falls within the data range. Common Result: The mean represents the central tendency and is often close to the dataset's central value.

# Calculate the median of the dataset

# Parameter: The dataset itself. Common Range: The median is within the data range. Common Result: The median represents the middle value, which is often close to the central value, especially in symmetric datasets.

# Calculate the mode of the dataset

# Parameter: The dataset itself. Common Range: The mode represents the most frequent value(s) in the dataset, which can vary. Common Result: The mode indicates the most prevalent category or value within the dataset.

# Calculate the variance of the dataset

# Parameter: The dataset itself. Common Range: The variance can vary widely based on data dispersion. Common Result: Low variance indicates data points are close to the mean, while high variance implies greater data spread.

# Calculate the standard deviation of the dataset

# Parameter: The dataset itself. Common Range: The standard deviation varies based on data dispersion. Common Result: A low standard deviation suggests data points are close to the mean, while a high standard deviation indicates greater spread.

# Calculate the correlation coefficient between two datasets

# Parameter: Two datasets being compared. Common Range: The correlation coefficient ranges from -1 to 1. Common Result: A high positive value (close to 1) indicates a strong positive linear relationship, while a high negative value (close to -1) indicates a strong negative linear relationship. A value close to 0 suggests a weak or no linear relationship.

# Perform a t-test to compare means of two datasets

# Parameter: Two datasets being compared. Common Range: The t-statistic varies, but a high value suggests significant differences in means. Common Result: A high t-statistic and a low p-value suggest that the means of the two datasets are significantly different, while a low t-statistic and a high p-value suggest no significant difference.

# Perform linear regression analysis

# Parameter: Two variables for regression analysis. Common Range: R-squared ranges from 0 to 1. Common Result: A high R-squared value (close to 1) indicates a strong linear relationship, meaning the dependent variable is well-explained by the independent variable(s).

# Perform chi-square test for independence

# Parameter: Contingency table of categorical variables. Common Range: The chi-square statistic varies, but a high value suggests dependence. Common Result: A high chi-square statistic and a low p-value indicate that the categorical variables are dependent, while a low statistic and a high p-value suggest independence.

# Perform ANOVA (Analysis of Variance)

# Parameter: Multiple datasets (more than two). Common Range: The F-statistic varies, but a high value suggests significant differences in means. Common Result: A high F-statistic and a low p-value indicate that at least one group differs significantly from the rest, while a low F-statistic and a high p-value suggest no significant differences.

# Perform Mann-Whitney U test for non-parametric data

# Parameter: Two independent datasets. Common Range: The U-statistic varies, but a low p-value suggests significant differences. Common Result: A low p-value indicates significant differences in distributions between the two groups.

# Perform Kruskal-Wallis H test for non-parametric data

# Parameter: Multiple datasets (more than two). Common Range: The H-statistic varies, but a low p-value suggests significant differences. Common Result: A low H-statistic and a low p-value indicate significant differences in medians among groups, while the opposite suggests no significant differences.

# Perform chi-square goodness-of-fit test

# Parameter: Observed and expected frequencies in categorical data. Common Range: The chi-square statistic varies. Common Result: A low chi-square statistic and a high p-value suggest that observed data fits the expected distribution well, while the opposite suggests a poor fit.

# Perform one-sample t-test

# Parameter: Sample data and a population mean. Common Range: The t-statistic varies. Common Result: A high t-statistic and a low p-value suggest that the sample mean is significantly different from the population mean, while the opposite suggests no significant difference.

# Perform paired t-test

# Parameter: Two related datasets (e.g., before and after measurements). Common Range: The t-statistic varies. Common Result: A high t-statistic and a low p-value indicate a significant difference between paired observations, while the opposite suggests no significant difference.

# Perform Pearson correlation test

# Parameter: Two continuous variables. Common Range: The correlation coefficient varies from -1 to 1. Common Result: A high positive correlation coefficient and a low p-value suggest a strong positive linear relationship, while a high negative correlation coefficient suggests a strong negative linear relationship.

# Perform Spearman rank correlation test

# Parameter: Two variables. Common Range: The correlation coefficient varies from -1 to 1. Common Result: A high positive correlation coefficient and a low p-value suggest a strong monotonic relationship, while a low p-value indicates a weak relationship or no relationship.

# MACHINE LEARNING:

## COMMON COMMANDS:

* Data
* Features
* Algorithms
* Learning types (Supervised, Unsupervised, Reinforcement, etc...)
* Model Training
* Evaluation Metrics
* Overfitting and Underfitting
* Model Selection
* Deployment

# Import necessary libraries

*import pandas as pd*

*import numpy as np*

*from sklearn.model\_selection import train\_test\_split*

*from sklearn.preprocessing import StandardScaler*

*from sklearn.linear\_model import LinearRegression*

*from sklearn.metrics import mean\_squared\_error, r2\_score*

# Load the dataset

# This command loads your dataset into a Pandas DataFrame for further analysis and modeling.

*data = pd.read\_csv('your\_dataset.csv')*

# Split the dataset into training and testing sets

# This command splits your dataset into two parts: one for training the model and one for testing its performance. The test\_size parameter determines the proportion of data used for testing.

*X = data.drop(columns=['target\_column'])*

*y = data['target\_column']*

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)*

# Standardize the features (optional)

# Standardization scales the features to have a mean of 0 and a standard deviation of 1. This is important for many machine learning algorithms that are sensitive to the scale of the features.

*scaler = StandardScaler()*

*X\_train = scaler.fit\_transform(X\_train)*

*X\_test = scaler.transform(X\_test)*

# Initialize and train a machine learning model

# You select a machine learning algorithm and initialize a model instance (e.g., LinearRegression) and then fit it to your training data to learn the underlying patterns.

*model = LinearRegression()*

*model.fit(X\_train, y\_train)*

# Make predictions on the test set

# You use the trained model to make predictions on the test data.

*y\_pred = model.predict(X\_test)*

# Evaluate the model's performance

# You calculate various metrics to assess how well the model performs on the test data.

mse = mean\_squared\_error(y\_test, y\_pred) # Mean Squared Error

*r2 = r2\_score(y\_test, y\_pred) # R-squared (Coefficient of Determination)*

# Display the model's performance metrics

# You print or visualize the evaluation metrics to assess how well the model has learned from the data.

*print(f"Mean Squared Error: {mse}")*

*print(f"R-squared (Coefficient of Determination): {r2}")*

## ADVANCED COMMANDS:

### Correlation:

To plot a correlation matrix, you first need to calculate the correlation coefficients between the features (or columns) of your dataset. You can use the pandas library to do this, as it has a built-in method called corr() that returns a correlation matrix as a DataFrame object. For example, if your dataset is stored in a variable called df, you can do:

# Calculate the correlation matrix

corr\_matrix = df.corr()

The corr() method by default uses the Pearson correlation coefficient, which measures the linear relationship between two variables. You can also specify other methods, such as Spearman or Kendall, by passing them as an argument to the method parameter.

Once you have the correlation matrix, you can plot it as a heatmap using the seaborn library, which is a visualization library based on matplotlib. The seaborn library has a function called heatmap() that takes a correlation matrix as input and plots a heatmap with color-coded cells that represent the correlation coefficients. The annot=True parameter adds the correlation coefficients to each cell, and the cmap parameter sets the color scheme for the heatmap. For example, you can do:

# Import seaborn

import seaborn as sns

# Plot the heatmap

sns.heatmap(corr\_matrix, annot=True, cmap="Blues")

This will produce a plot similar to the one on page 13 of the PDF document. You can customize the plot further by changing the parameters of the heatmap() function or by using matplotlib functions to adjust the figure size, labels, title, etc.

To plot a partial correlation matrix, which measures the correlation between two variables after controlling for other variables, you need to use a different library, such as pingouin or statsmodels. These libraries have functions that can calculate and plot partial correlation matrices. For example, using pingouin, you can do:

# Import pingouin

import pingouin as pg

# Calculate and plot the partial correlation matrix

pg.plotting.plot\_partial\_corr(data=df)

This will produce a plot similar to the one on page 14 of the PDF document. You can also customize this plot by changing the parameters of the plot\_partial\_corr() function.

To plot a scatterplot matrix, which shows pairwise scatterplots between features along with histograms or density plots on the diagonal, you can use either seaborn or pandas. Both libraries have functions that can create scatterplot matrices from DataFrames. For example, using seaborn, you can do:

# Import seaborn

import seaborn as sns

# Plot the scatterplot matrix

sns.pairplot(data=df)

This will produce a plot similar to the one on page 15 of the PDF document. You can also customize this plot by changing the parameters of the pairplot() function or by using matplotlib functions to adjust the figure aesthetics.

### Correlation (is the degree to which things move together (relationship))

#that describes the extent to which two columns change together

# looks at strenght and direction

​

data['Y'].corr(data['BMI'])

0.5864501344746887

data['BMI'].corr(data['AGE'])

0.18508466614655553

data.corr()

|  | **AGE** | **SEX** | **BMI** | **BP** | **S1** | **S2** | **S3** | **S4** | **S5** | **S6** | **Y** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **AGE** | 1.000000 | 0.173737 | 0.185085 | 0.335428 | 0.260061 | 0.219243 | -0.075181 | 0.203841 | 0.270774 | 0.301731 | 0.187889 |
| **SEX** | 0.173737 | 1.000000 | 0.088161 | 0.241010 | 0.035277 | 0.142637 | -0.379090 | 0.332115 | 0.149916 | 0.208133 | 0.043062 |
| **BMI** | 0.185085 | 0.088161 | 1.000000 | 0.395411 | 0.249777 | 0.261170 | -0.366811 | 0.413807 | 0.446157 | 0.388680 | 0.586450 |
| **BP** | 0.335428 | 0.241010 | 0.395411 | 1.000000 | 0.242464 | 0.185548 | -0.178762 | 0.257650 | 0.393480 | 0.390430 | 0.441482 |
| **S1** | 0.260061 | 0.035277 | 0.249777 | 0.242464 | 1.000000 | 0.896663 | 0.051519 | 0.542207 | 0.515503 | 0.325717 | 0.212022 |
| **S2** | 0.219243 | 0.142637 | 0.261170 | 0.185548 | 0.896663 | 1.000000 | -0.196455 | 0.659817 | 0.318357 | 0.290600 | 0.174054 |
| **S3** | -0.075181 | -0.379090 | -0.366811 | -0.178762 | 0.051519 | -0.196455 | 1.000000 | -0.738493 | -0.398577 | -0.273697 | -0.394789 |
| **S4** | 0.203841 | 0.332115 | 0.413807 | 0.257650 | 0.542207 | 0.659817 | -0.738493 | 1.000000 | 0.617859 | 0.417212 | 0.430453 |
| **S5** | 0.270774 | 0.149916 | 0.446157 | 0.393480 | 0.515503 | 0.318357 | -0.398577 | 0.617859 | 1.000000 | 0.464669 | 0.565883 |
| **S6** | 0.301731 | 0.208133 | 0.388680 | 0.390430 | 0.325717 | 0.290600 | -0.273697 | 0.417212 | 0.464669 | 1.000000 | 0.382483 |
| **Y** | 0.187889 | 0.043062 | 0.586450 | 0.441482 | 0.212022 | 0.174054 | -0.394789 | 0.430453 | 0.565883 | 0.382483 | 1.000000 |

# Needs this to make the heatmap

mask = np.zeros\_like(data.corr())

triangle\_indices = np.triu\_indices\_from(mask)

mask[triangle\_indices] = True

mask

array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],

[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1.],

[0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],

[0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],

[0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],

[0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])

#The annot=True argument means that the correlation coefficients will be written on the heatmap

#, and annot\_kws={'size': 14} sets the font size of these annotations to 14

​

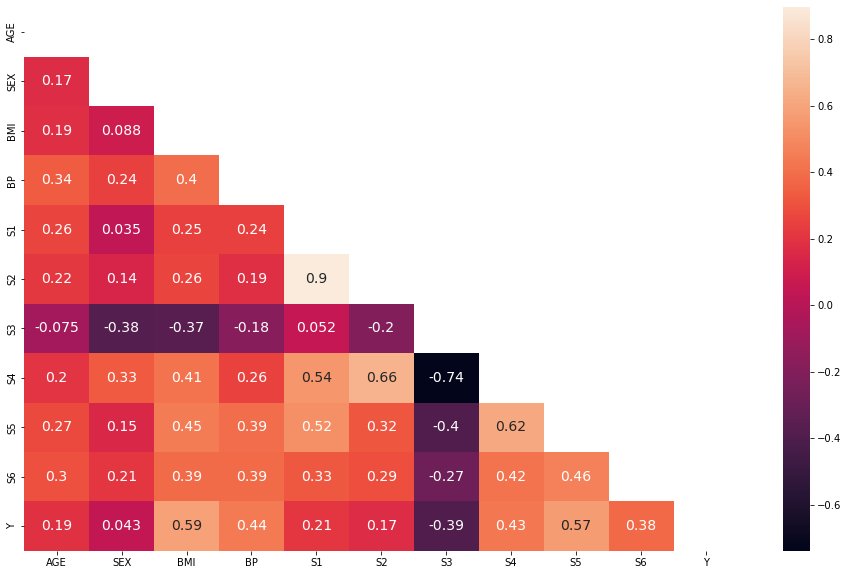
plt.figure(figsize=(16, 10))

sns.heatmap(data.corr(), mask=mask, annot=True, annot\_kws={'size': 14}) # {'size': 14} is a python dictionary (key/value pair)

plt.xticks(fontsize=10)

plt.yticks(fontsize=10)

plt.show()



# Scatter plot between S4 and S3 Correlation

s3\_s4\_corr = round(data['S3'].corr(data['S4']), 3)

plt.figure(figsize=(9, 6))

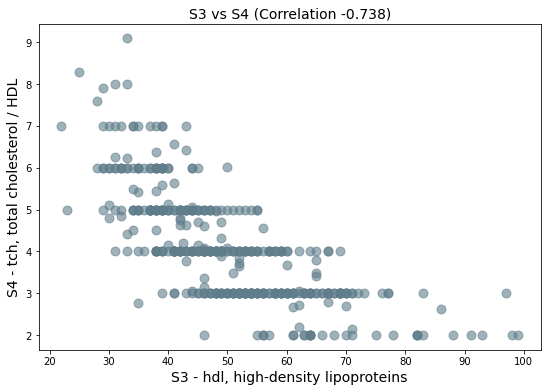
plt.scatter(x=data['S3'], y=data['S4'], alpha=0.6, s=80, color='#607D8B')

plt.title(f'S3 vs S4 (Correlation {s3\_s4\_corr})', fontsize=14)

plt.xlabel('S3 - hdl, high-density lipoproteins', fontsize=14)

plt.ylabel('S4 - tch, total cholesterol / HDL', fontsize=14)

plt.show()



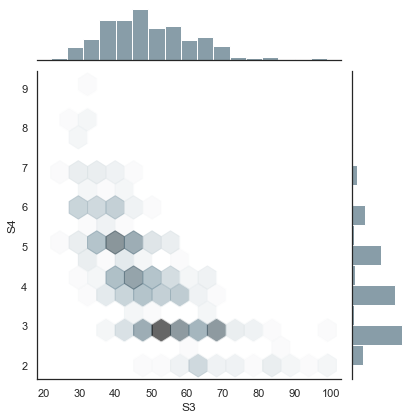
sns.set()

sns.set\_context('notebook')

sns.set\_style('white')

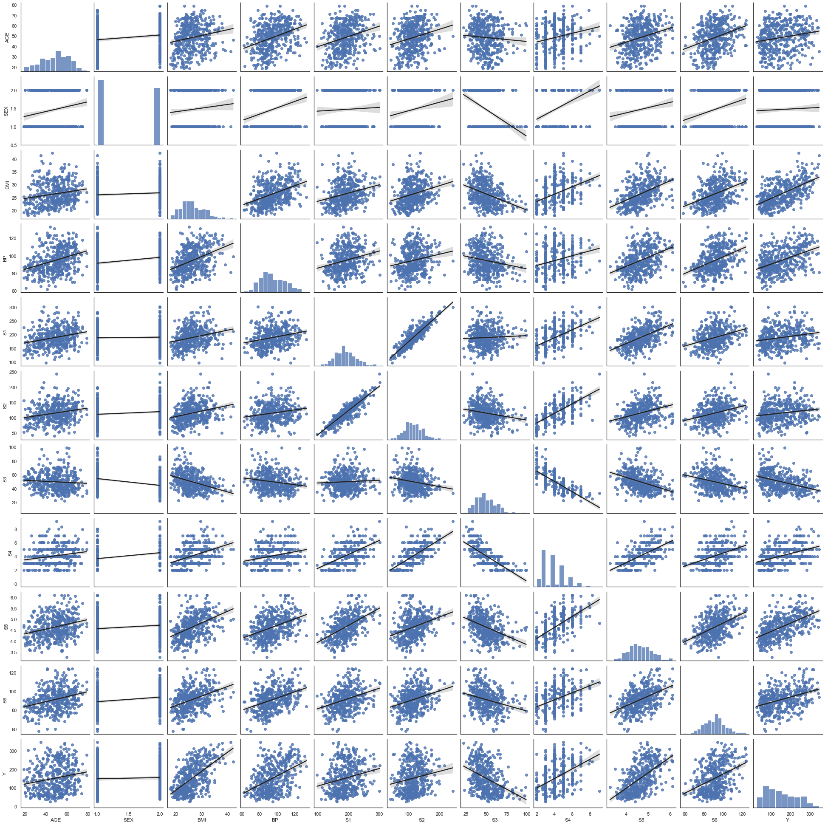
sns.jointplot(x=data['S3'], y=data['S4'], height=6, color='#607D8B', kind='hex', joint\_kws={'alpha': 0.6})

plt.show()



sns.pairplot(data, kind='reg', plot\_kws={'line\_kws':{'color':'#212121'}})

plt.show()



### Training & Test Dataset Split

responses = data['Y']

features = data.drop('Y', axis=1)

​

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, responses,

test\_size=0.2, random\_state=30)

​

#calculating the proportion of the total data that is being used for training.

#This should return 0.8, since you’re using 80% of the data for training (test\_size=0.2).

#This is a good way to verify that your data has been split correctly.

len(X\_train)/len(features)

0.7986425339366516

len(X\_test)/len(features)

0.20135746606334842

### Regression

regr = LinearRegression()

regr.fit(X\_train, y\_train)

#multiy varieble

print('Training data r-squared:', regr.score(X\_train, y\_train))

print('Test data r-squared:', regr.score(X\_test, y\_test))

​

print('Intercept', regr.intercept\_)

pd.DataFrame(data=regr.coef\_, index=X\_train.columns, columns=['coef'])

#The coefficients represent the change in the response variable for a one unit change in

#the corresponding feature, assuming all other features are held constant.

Training data r-squared: 0.5264783626678893

Test data r-squared: 0.4653044632644133

Intercept -368.2185304349134

|  | **coef** |
| --- | --- |
| **AGE** | 0.029063 |
| **SEX** | -23.059093 |
| **BMI** | 5.602677 |
| **BP** | 1.254404 |
| **S1** | -1.334738 |
| **S2** | 0.955752 |
| **S3** | 0.588029 |
| **S4** | 3.158733 |
| **S5** | 78.918445 |
| **S6** | 0.226053 |

The R-squared values indicate how well your model fits your data. The R-squared value for the training data is approximately 0.526, which means that about 52.6% of the variance in the target variable can be explained by the features in your model when it’s trained on the training data.

The R-squared value for the test data is approximately 0.465, which means that about 46.5% of the variance in the target variable can be explained by the features in your model when it’s tested on the test data. This lower R-squared value for the test data suggests that your model may not generalize as well to unseen data as it does to the data it was trained on.

The intercept of your model is approximately -368.22. This means that if all of your feature values were zero, your model would predict a value of -368.22 for the target variable. However, this interpretation might not make sense depending on the context and scale of your features and target variable.

-AGE: For each one unit increase in AGE, the target variable increases by 0.029063, assuming all other variables stay the same. -SEX: For each one unit increase in SEX, the target variable decreases by 23.059093, assuming all other variables stay the same. -BMI: For each one unit increase in BMI, the target variable increases by 5.602677, assuming all other variables stay the same. -BP: For each one unit increase in BP, the target variable increases by 1.254404, assuming all other variables stay the same. -S1: For each one unit increase in S1, the target variable decreases by 1.334738, assuming all other variables stay the same. -S2: For each one unit increase in S2, the target variable increases by 0.955752, assuming all other variables stay the same. -S3: For each one unit increase in S3, the target variable increases by 0.588029, assuming all other variables stay the same. -S4: For each one unit increase in S4, the target variable increases by 3.158733, assuming all other variables stay the same. -S5: For each one unit increase in S5, the target variable increases by 78.918445, assuming all other variables stay the same. -S6: For each one unit increase in S6, the target variable increases by 0.226053, assuming all other variables stay the same.

These interpretations assume that your model is a good fit for your data and that there are no significant interactions between your features that aren’t accounted for in your model. It’s also important to note that these interpretations are based on a linear relationship between each feature and the target variable.

close to zero under fitting closeto 1 over fitting try to applie adjusted R square

### Data Transformations

data['Y'].skew()

0.44056293407014124

In simpler terms, skewness tells you the amount and direction of skew (departure from horizontal symmetry). The skewness value can be positive or negative, or even undefined. If skewness is less than -1 or greater than 1, the distribution is highly skewed. If skewness is between -1 and -0.5 or between 0.5 and 1, the distribution is moderately skewed. If skewness is between -0.5 and 0.5, the distribution is approximately symmetric.

So, data['Y'].skew() will return the skewness of the values in column ‘Y’ of your dataframe ‘data’. If you’ve already calculated this and got 0.44056293407014124, it means that your data in ‘Y’ is approximately symmetric with a slight right skew, as it’s between 0.5 and -0.5. Positive=right Negative=left

y\_log = np.log(data['Y'])

y\_log.tail()

437 5.181784

438 4.644391

439 4.882802

440 5.393628

441 4.043051

Name: Y, dtype: float64

y\_log.skew()

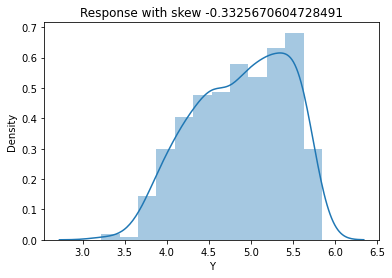
-0.3325670604728491

A negative skewness value indicates that the distribution is skewed to the left. In your case, a skewness of -0.3325670604728491 means that your data is approximately symmetric but with a slight left skew.

sns.distplot(y\_log)#rember this is PDF line comes with

plt.title(f'Response with skew {y\_log.skew()}')

plt.show()



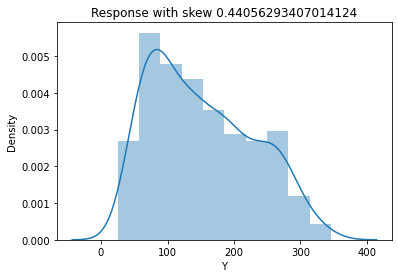
tmpdata = data['Y']

# before log

sns.distplot(tmpdata)

plt.title(f'Response with skew {tmpdata.skew()}')

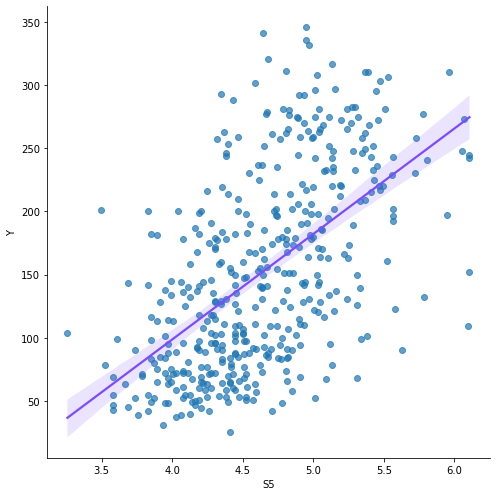
plt.show()



# before transformed Log Data

sns.lmplot(x='S5', y='Y', data=data, height=7, scatter\_kws={'alpha': 0.7}, line\_kws={'color':'#7C4DFF'})

plt.show()



# after transformed Log Data

#scatter plot with a line of best fit (also known as a regression line).

transformed\_data = features

transformed\_data['LOG\_RESPONSE'] = y\_log

sns.lmplot(x='S5', y='LOG\_RESPONSE', data=transformed\_data, height=7, scatter\_kws={'alpha': 0.7}, line\_kws={'color':'#5D4037'})

plt.show()

### Mean Squared Error (MSE):

The Mean Squared Error (MSE) is a measure of how close a fitted line is to data points. It does this by taking the distances from the points to the regression line (these distances are the “errors”) and squaring them. The squaring is necessary to remove any negative signs. It also gives more weight to larger differences. It’s called the mean squared error as you’re finding the average of a set of errors.

# imports:

*from sklearn.metrics import mean\_squared\_error*

*from sklearn.linear\_model import LinearRegression*

*from sklearn.model\_selection import train\_test\_split*

*from sklearn.datasets import make\_regression*

*import numpy as np*

# Generate a random regression problem

*X, y = make\_regression(n\_samples=100, n\_features=1, noise=0.1)*

# Split the data into training/testing sets

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)*

# Create linear regression object

*regr = LinearRegression()*

# Train the model using the training sets

*regr.fit(X\_train, y\_train)*

# Make predictions using the testing set

*y\_pred = regr.predict(X\_test)*

# The mean squared error

*print('Mean squared error: %.2f' % mean\_squared\_error(y\_test, y\_pred))*

In this code:

* We first import the necessary libraries.
* We generate a random regression problem using make\_regression.
* We split our dataset into a training set and a test set.
* We create a LinearRegression object.
* We train our model using our training data.
* We then predict the output for our test dataset using the trained model.
* Finally, we calculate the MSE using the mean\_squared\_error function from sklearn.metrics, which computes the mean squared error between y\_test (the actual outputs) and y\_pred (the predicted outputs).

The result you get from this will be a non-negative floating point number. **The closer to 0 this is, the better your model’s predictions match the true values.** A value of 0 indicates that your model’s predictions perfectly match the true values, but this is typically not achievable with real-world data.

### P values & Evaluating Coefficients

#This line adds a column of ones to your training features, which represents the intercept term in your regression model.

X\_incl\_const = sm.add\_constant(X\_train)

model = sm.OLS(y\_train, X\_incl\_const)

results = model.fit()​

pd.DataFrame({'coef': results.params, 'p-value': round(results.pvalues, 3)})

#P ≤ 0,05 is considered statistically significant.

### BIC

BIC is a model selection criterion for a finite list of models. A way to measure complexity between models. The lower the BIC value the better the model (a lower value is better).

import numpy as np

import statsmodels.api as sm

​

# Generate some example data

np.random.seed(0)

n\_samples = 100

X = np.random.randn(n\_samples, 2)

y = 2 \* X[:, 0] + X[:, 1] + np.random.randn(n\_samples)

print (X)

print(y)

print(X\_incl\_const)

​

# Add a constant to the features for the intercept term

X\_incl\_const = sm.add\_constant(X)

​

# Fit an OLS model

model = sm.OLS(y, X\_incl\_const)

results = model.fit()

​

# Print the BIC

print('BIC: ', results.bic)

The Bayesian Information Criterion (BIC) is a way to compare the goodness of fit of different statistical models applied to the same data set. It takes into account both the complexity of the model (i.e., the number of parameters) and the likelihood of the data given the model.

The BIC provides a balance between model complexity and model performance. A model with more parameters may fit the data better, but it’s also more complex and may overfit the data. The BIC helps us choose the simplest model that still performs well.

In simple terms, lower BIC values indicate better models. So when comparing models, the one with the lowest BIC is generally preferred.

### Variance Inflation Factor (VIF)

We are testing for Multicollinearity. VIF is a measure of collinearity amongst the features within a regression model. A high VIF value indicates that a feature is highly correlated with other features, potentially leading to unstable coefficient estimates.

import pandas as pd

import numpy as np

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

​

# Assuming you have a DataFrame 'df' with your data

df = pd.DataFrame(

np.array([

[2.1, 2.5, 3.3, 2.7],

[3.2, 3.5, 3.7, 3.1],

[2.2, 2.7, 3.1, 2.9],

[2.8, 3.4, 3.7, 3.0],

[3.0, 3.2, 3.5, 3.6]]),

columns=['Feature1', 'Feature2', 'Feature3', 'Feature4'])

​

# Add a constant for the intercept term

X = sm.add\_constant(df)

​

# Calculate VIF for each feature

vif = pd.DataFrame()

vif["features"] = X.columns

vif["VIF"] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]

​

print(vif)

​

features VIF

0 const 1055.399734

1 Feature1 22.081770

2 Feature2 15.380944

3 Feature3 6.813469

4 Feature4 3.800369

A VIF of 1 means that there is no correlation among the i-th variable and the remaining variables, VIF between 1 and 5 suggests moderate correlation but is usually not of concern, and a value greater than 5 indicates high correlation and suggests that the variable is redundant for prediction because it’s highly correlated with other variables.

# OTHER:

To search for a patient with 3 sicknesses in a DataFrame, you would first need to have a DataFrame that contains this information. For example, if you have a DataFrame df where each row represents a patient and there is a column ‘sicknesses’ that contains the number of sicknesses for each patient, you can use the following code:

# This will give you a new DataFrame with only the patients who have 3 sicknesses

*df\_three\_sicknesses = df[df['sicknesses'] == 3]*

To group data in a DataFrame, you can use the groupby function. For example, if you wanted to group your data by the number of sicknesses and find the average age of patients within each group, you could do:

# This assumes there is an 'age' column in your DataFrame

*average\_age\_by\_sickness = df.groupby('sicknesses')['age'].mean()*

This will give you a new DataFrame where the index is the number of sicknesses and the ‘age’ column contains the average age of patients with that number of sicknesses.

Remember to replace ‘sicknesses’ and ‘age’ with the actual column names in your DataFrame. If your DataFrame is structured differently, you may need to adjust this code to fit your specific situation. If you provide more details about your DataFrame and what exactly you’re trying to accomplish, I could give more specific advice.

\_\_\_\_\_\_\_\_

If you have a DataFrame where each row represents an instance of a patient having a sickness, you can count the number of unique sicknesses for each patient and then filter for patients with exactly 3 unique sicknesses. Here’s how you can do it:

# This assumes 'PatientId' is the column with the patient's ID and 'Sickness' is the column with the sicknesses

*sickness\_counts = df.groupby('PatientId')['Sickness'].nunique()*

# This will give you a DataFrame with only the patients who have exactly 3 unique sicknesses

*patients\_with\_three\_sicknesses = sickness\_counts[sickness\_counts == 3]*

In this code:

* We first group by ‘PatientId’ and then use the nunique function to count the number of unique ‘Sickness’ for each patient.
* We then filter sickness\_counts to get only those patients who have exactly 3 unique sicknesses.

The resulting patients\_with\_three\_sicknesses will be a pandas Series where the index is the patient IDs and the values are all 3. If you want this as a list of patient IDs, you can do patients\_with\_three\_sicknesses.index.tolist().

Remember to replace ‘PatientId’ and ‘Sickness’ with the actual column names in your DataFrame. If your DataFrame is structured differently, you may need to adjust this code to fit your specific situation. If you provide more details about your DataFrame and what exactly you’re trying to accomplish, I could give more specific advice.

 common pandas DataFrame operations that you might find useful:

1. **Grouping**: The groupby function is used to split the data into groups based on some criteria. The syntax is df.groupby('column\_name').
2. **Aggregation**: This is often used with groupby. You can compute group summary statistics like count, sum, mean, median, etc. For example, df.groupby('column\_name').mean() computes the mean value for each group.
3. **Filtering**: You can select a subset of rows in a DataFrame based on some condition. For example, df[df['column\_name'] > 0] selects only the rows where ‘column\_name’ is greater than 0.
4. **Transformation**: You can perform some operation on the groups and return a like-indexed object. For example, df.groupby('column\_name').transform(lambda x: (x - x.mean()) / x.std()) standardizes the data (zero mean and unit variance) within each group.
5. **Applying**: You can apply a function to each group of values. For example, df.groupby('column\_name').apply(np.sum) applies the numpy sum function to each group.
6. **Pivoting**: The pivot\_table function is used to create a new derived table out of a given one. For example, df.pivot\_table(index='column1', columns='column2', values='column3', aggfunc=np.sum) creates a pivot table that groups by ‘column1’ and ‘column2’ and sums ‘column3’.
7. **Merging/Joining**: You can combine DataFrames using a similar logic as merging SQL Tables together. For example, pd.merge(df1, df2, on='column\_name') merges df1 and df2 on ‘column\_name’.
8. **Concatenating**: This is used to combine DataFrames along a particular axis with optional set logic along the other axes. For example, pd.concat([df1, df2]) concatenates df1 and df2 along the row axis.
9. **Distinct**: To get distinct rows in DataFrame, you can use df.drop\_duplicates(). If you want to get distinct values of a column, you can use df['column\_name'].unique().
10. **Sorting**: You can sort a DataFrame based on one or more columns. For example, df.sort\_values(by='column\_name') sorts by ‘column\_name’.

Remember to replace ‘df’, ‘df1’, ‘df2’, ‘column\_name’, ‘column1’, ‘column2’, and ‘column3’ with your actual DataFrame variable names and column names.

### EXAMPLES:

1. **Grouping:**

The groupby function is used to split the data into groups based on some criteria.

import pandas as pd

# Create a DataFrame

df = pd.DataFrame({

'PatientId': ['1', '2', '1', '3', '2', '1', '3', '3'],

'Sickness': ['Flu', 'Cold', 'COVID-19', 'Flu', 'Flu', 'Cold', 'COVID-19', 'Cold'],

'Price': [10, 20, 30, 40, 50, 60, 70, 80] })

# Group by column 'PatientId'

grouped = df.groupby('PatientId')

print(grouped.sum())

This will print the total price of medicine bought by each patient.

1. **Aggregation:**

This is often used with groupby. You can compute group summary statistics like count, sum, mean, median, etc.

# Using the same DataFrame as above

# Group by column 'PatientId' and calculate the mean of the prices

print(df.groupby('PatientId')['Price'].mean())

This will print the average price of medicine bought by each patient.

1. **Filtering:**

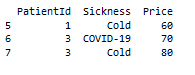
You can select a subset of rows in a DataFrame based on some condition.

# Using the same DataFrame as above

# Select rows where column 'Price' is greater than 50

print(df[df['Price'] > 50])

This will print the rows where the price of the medicine is greater than 50.

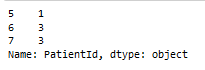


\*\*If you only want to display the ‘PatientId’ and not the whole row where the ‘Price’ is greater than 50, you can do so by selecting the ‘PatientId’ column after the condition. Here’s how you can do it:

# Select 'PatientId' where 'Price' is greater than 50

print(df[df['Price'] > 50]['PatientId'])

This will print only the ‘PatientId’ where the price of the medicine is greater than 50.



\*\*# Select 'PatientId' where 'Price' is greater than 50 and group by 'PatientId'

patient\_ids = df[df['Price'] > 50]['PatientId'].groupby(df['PatientId']).apply(list).reset\_index(name='PatientIds')

print(patient\_ids)

#This will print a DataFrame with unique ‘PatientId’ who paid more than 50 at least once, and a list of their ‘PatientId’ repeated as many times as they paid more than 50.



\*\* If you only want to see the unique ‘PatientId’ for those who paid more than 50, you can do so by using a combination of boolean indexing and the unique() function. Here’s how you can do it:

# Select 'PatientId' where 'Price' is greater than 50 and get unique values

unique\_patient\_ids = df[df['Price'] > 50]['PatientId'].unique()

print(unique\_patient\_ids)

This will print an array with the unique ‘PatientId’ who paid more than 50 at least once.



1. **Top/Bottom three Patients:**

To find out which three patients paid the most, you can use the groupby function to group the data by ‘PatientName’, then use the sum function to calculate the total price paid by each patient, and finally use the nlargest function to get the three patients who paid the most.

import pandas as pd

# Create a DataFrame

df = pd.DataFrame({

'PatientName': ['John', 'Mary', 'John', 'Mary', 'John', 'Bob', 'Bob', 'Mary'],

'Sickness': ['Flu', 'Cold', 'COVID-19', 'Flu', 'Flu', 'Cold', 'COVID-19', 'Cold'],

'Price': [10, 20, 30, 40, 50, 60, 70, 80] })

# Group by 'PatientName', calculate the total price paid by each patient,

# and get the three patients who paid the most

top\_three\_patients = df.groupby('PatientName')['Price'].sum().nlargest(3)

print(top\_three\_patients)

This will print a Series with the names of the top three patients who paid the most and the total amount they paid. The patients are sorted in descending order of the total amount paid.

**To arrange the total price paid by each patient in ascending order, you can use** the sort\_values function. Here’s how you can do it:

# Group by 'PatientName', calculate the total price paid by each patient,

# and sort in ascending order

sorted\_patients = df.groupby('PatientName')['Price'].sum().sort\_values()

print(sorted\_patients)

This will print a Series with the names of the patients and the total amount they paid, sorted in ascending order of the total amount paid.

To get the three patients who paid the least, you can use the nsmallest function.

**To sort the DataFrame by the ‘Price’ column in descending order, you can set** the ascending parameter of the sort\_values function to False. Here’s how you can do it:

# Sort by column 'Price' in descending order

print(df.sort\_values(by='Price', ascending=False))

This will print the DataFrame sorted by the price of the medicine in descending order.

# Group by 'PatientName', calculate the total price paid by each patient,

# and get the three patients who paid the least

bottom\_three\_patients = df.groupby('PatientName')['Price'].sum().nsmallest(3)

print(bottom\_three\_patients)

This will print a Series with the names of the bottom three patients who paid the least and the total amount they paid. The patients are sorted in ascending order of the total amount paid.

1. **Distinct:**

To get distinct rows in DataFrame, you can use df.drop\_duplicates(). If you want to get distinct values of a column, you can use df['column\_name'].unique().

# Using the same DataFrame as above

# Get distinct sicknesses

print(df['Sickness'].unique())

This will print the unique sicknesses in the DataFrame.

1. **Sorting:**

You can sort a DataFrame based on one or more columns.

# Using the same DataFrame as above

# Sort by column 'Price'

print(df.sort\_values(by='Price'))

This will print the DataFrame sorted by the price of the medicine.

Remember to replace 'df', 'PatientId', 'Sickness', and 'Price' with your actual DataFrame variable names and column names.

### Unique Distict:

Sure, here’s how you can get unique or distinct values in a DataFrame:

The unique() function is used to find the unique elements of an array. It returns the sorted unique elements of an array.

import pandas as pd

# Create a DataFrame

df = pd.DataFrame({

'PatientId': ['1', '2', '1', '3', '2', '1', '3', '3'],

'Sickness': ['Flu', 'Cold', 'COVID-19', 'Flu', 'Flu', 'Cold', 'COVID-19', 'Cold'],

'Price': [10, 20, 30, 40, 50, 60, 70, 80] })

# Get unique sicknesses

unique\_sicknesses = df['Sickness'].unique()

print(unique\_sicknesses)

This will print: ['Flu' 'Cold' 'COVID-19']

The nunique() function is used to count distinct observations over requested axis. It returns the number of distinct observations.

# Count unique sicknesses

num\_unique\_sicknesses = df['Sickness'].nunique()

print(num\_unique\_sicknesses)

This will print: 3

The drop\_duplicates() function is used to remove duplicates from the DataFrame.

# Drop duplicate rows

df\_no\_duplicates = df.drop\_duplicates()

print(df\_no\_duplicates)

This will print a DataFrame with no duplicate rows.

## ADVANCED COMBOS:

**To find out the names of patients who had the ‘Flu’ sickness in ‘South Africa’ and paid more than 50 for their medicine**

To find the Patient names who had the sickness of 'flu' in the country 'South Africa' and paid more than 50 for their medicine in a Python Jupyter Notebook, you can use the pandas library for data manipulation. Here's the code with an explanation and an applied example:

python

import pandas as pd

# Sample DataFrame (replace this with your actual DataFrame)

data = {

'Patient\_Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],

'Sickness': ['flu', 'fever', 'flu', 'cough', 'flu'],

'Country': ['South Africa', 'USA', 'South Africa', 'Canada', 'South Africa'],

'Price\_Paid': [60, 45, 70, 30, 55]

}

df = pd.DataFrame(data)

# Filter the DataFrame based on the criteria

condition = (df['Sickness'] == 'flu') & (df['Country'] == 'South Africa') & (df['Price\_Paid'] > 50)

result = df[condition]['Patient\_Name']

# Display the result

print(result)

Explanation:

1. We import the pandas library and create a sample DataFrame df with columns representing Patient names, Sickness, Country, and Price paid for medicine. Replace this with your actual DataFrame.
2. We create a condition condition that checks three conditions:
   * df['Sickness'] == 'flu' checks if the sickness is 'flu'.
   * df['Country'] == 'South Africa' checks if the country is 'South Africa'.
   * df['Price\_Paid'] > 50 checks if the price paid is more than 50.
3. We filter the DataFrame using the condition and extract the 'Patient\_Name' column for the patients who meet all three conditions. This gives us a Series containing the names of patients who had 'flu' in 'South Africa' and paid more than 50 for their medicine.
4. Finally, we print the result, which contains the names of patients who meet the criteria.

Example: Suppose you have a medical dataset with patient information, including their names, sicknesses, countries, and prices paid for medicine. Using the provided code with your actual dataset, you can find the patients who had 'flu' in 'South Africa' and paid more than 50 for their medicine. Here's an example result:

vbnet

2 Charlie

4 Eve

Name: Patient\_Name, dtype: object

In this example, the code identifies two patients, Charlie and Eve, who had 'flu' in 'South Africa' and paid more than 50 for their medicine based on the given criteria.