# Data Visualization

--

## Step 1: Required `pip install` Commands

Before using the libraries, ensure they are installed in your environment. Open your terminal or Jupyter notebook and run these commands:

```
pip install numpy
pip install pandas
pip install matplotlib
pip install seaborn
pip install plotly
pip install scikit-learn
pip install statsmodels
pip install scipy
pip install umap-learn
pip install bokeh
pip install missingno
pip install xgboost
pip install category_encoders
```

If you're working in a **Jupyter Notebook**, you can run the commands with `!` at the beginning:

```
!pip install numpy pandas matplotlib seaborn plotly scikit-learn statsmodels
scipy umap-learn bokeh missingno xgboost category_encoders
```

---

## Step 2: Import Statements for Data Analysis, Visualization, Scaling, and Cleaning

Here’s a breakdown of all the import statements you'll need for your data preparation and visualization tasks:

### General Data Analysis & Preprocessing

```
import numpy as np
```

```
import pandas as pd
```

## Data Cleaning & Handling Missing Values

```
import missingno as msno
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from category_encoders import TargetEncoder, OrdinalEncoder
```

## Data Scaling & Normalization

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.preprocessing import Normalizer, PowerTransformer
```

## Data Transformation

```
from sklearn.preprocessing import FunctionTransformer, PolynomialFeatures
from scipy.stats import boxcox
```

## Visualization Libraries

```
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from bokeh.plotting import figure, show
from bokeh.io import output_notebook
import umap
```

## Advanced Statistical Analysis & Plotting

```
import statsmodels.api as sm
from statsmodels.graphics.mosaicplot import mosaic
from scipy import stats
from pandas.plotting import scatter_matrix, parallel_coordinates
```

## Machine Learning Models & Pipelines

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
import xgboost as xgb
```

### Exploratory Data Analysis & Correlation

```
from pandas.plotting import autocorrelation_plot
from pandas.plotting import lag_plot
import seaborn as sns
```

---

## Step 3: Explanation of Key Packages

- `numpy` & `pandas`: Essential for numerical computations and data manipulation.
- `missingno`: Useful for visualizing missing data.
- `scikit-learn`: A rich library for machine learning, scaling, preprocessing, and model evaluation.
- `matplotlib` & `seaborn`: Great for static visualizations and plots.
- `plotly` & `bokeh`: Perfect for interactive and dynamic visualizations.
- `statsmodels`: Used for statistical analysis, hypothesis testing, and regression models.
- `category_encoders`: Provides various encoding techniques like TargetEncoder and OrdinalEncoder.
- `umap`: Used for dimensionality reduction similar to t-SNE, but often faster and more scalable.
- `xgboost`: A popular gradient boosting library for classification and regression.

---

## Step 4: Common Custom Functions for Data Cleaning and Visualization

You can also create custom functions using these imports to streamline your workflow:

### Example: Function to Clean and Scale Data

```
def clean_and_scale_data(df):
    # Remove duplicates
    df.drop_duplicates(inplace=True)

    # Handle missing values
    imputer = SimpleImputer(strategy='mean')
    df.fillna(imputer.fit_transform(df.select_dtypes(include=['float64',
'int64'])))

    # Scale numerical data
    scaler = StandardScaler()
    df[df.select_dtypes(include=['float64', 'int64']).columns] =
scaler.fit_transform(
        df.select_dtypes(include=['float64', 'int64']))

    return df
```

---

This comprehensive list should cover all your needs for data visualization, cleaning, scaling, and transforming data for machine learning. Let me know if you need more specific examples or explanations for any of the libraries!

# 📊 Comprehensive Data Visualization Guide Using the UGRansome Dataset

## 📝 1. Introduction

Data visualization is essential for understanding patterns, relationships, and trends in your data. It helps identify anomalies, outliers, and correlations that can improve your analysis or model performance. In this guide, we will cover the following types of visualizations:

- Line graphs
- Bar charts
- Histograms
- Density plots
- Scatter plots
- Correlation heatmaps
- Box plots
- Pair plots

We will use your **UGRansome dataset** as a use case to illustrate these concepts.

---

## 📈 2. Line Graphs

Line graphs are great for visualizing trends over time or any continuous variable. In the context of your dataset, we can use a line graph to visualize how the **Netflow Bytes** or **BTC** values change over time.

### Example: Line Graph of 'Netflow_Bytes' Over Time

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(df2['Time'], df2['Netflow_Bytes'], color='blue')
plt.title('Netflow Bytes Over Time')
plt.xlabel('Time')
plt.ylabel('Netflow Bytes')
plt.grid(True)
plt.show()
```

**What This Shows**:

- Helps identify peaks in network traffic over time, which could indicate attacks or suspicious activity.

**When to Use**:

- For time-series data or when analyzing trends over a continuous variable.

---

## 📊 3. Bar Charts

Bar charts are useful for comparing categories. In your dataset, categorical variables like **Protocol**, **Flag**, **Family**, or **Clusters** can be visualized using bar charts.

### Example: Bar Chart of 'Protocol' Counts

```
import seaborn as sns

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

```
ax = sns.countplot(x='Protocol', data=df2)
plt.title('Count of Different Protocols')
plt.xlabel('Protocol')
plt.ylabel('Count')

# Add count labels on top of bars
for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}', (p.get_x() + p.get_width() / 2,
p.get_height()),
                ha='center', va='bottom', fontsize=10)
plt.xticks(rotation=45)
plt.show()
```

**What This Shows**:

- Allows you to see which protocols are most frequently used, which can help identify abnormal traffic.

**When to Use**:

- For categorical data where you want to compare the frequency of different categories.

---

## 📉 4. Histograms

Histograms are used to visualize the distribution of numerical data. In your dataset, we can plot histograms for **BTC**, **USD**, or **Netflow_Bytes**.

## Example: Histogram of 'BTC' Values

```
plt.figure(figsize=(10, 6))
plt.hist(df2['BTC'], bins=30, color='skyblue', edgecolor='black')
plt.title('Distribution of BTC Values')
plt.xlabel('BTC')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

**What This Shows**:

- Provides insights into the distribution of Bitcoin transactions, which could indicate suspicious spikes in usage.

**When to Use**:

- To understand the frequency distribution of numerical data.

---

## 🌐 5. Density Plots (Kernel Density Estimation)

Density plots are useful for estimating the probability density function of a continuous variable.

### Example: Density Plot of 'USD'

```
sns.kdeplot(df2['USD'], shade=True, color='green')
plt.title('Density Plot of USD Values')
plt.xlabel('USD')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```

**What This Shows**:

- Useful for understanding the distribution shape and detecting multimodal distributions.

**When to Use**:

- When you want to visualize the distribution of data while smoothing out noise.

---

## 📏 6. Box Plots

Box plots are ideal for detecting outliers and visualizing the spread of data.

### Example: Box Plot of 'Netflow_Bytes'

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='Clusters', y='Netflow_Bytes', data=df2)
plt.title('Box Plot of Netflow Bytes by Cluster')
plt.xlabel('Clusters')
plt.ylabel('Netflow Bytes')
plt.grid(True)
plt.show()
```

**What This Shows**:

- Useful for comparing the distribution of network bytes across different clusters to detect anomalies.

**When to Use**:

- For identifying outliers and understanding the variability within groups.

---

## 🔄 7. Scatter Plots

Scatter plots are used to explore relationships between two numerical variables.

### Example: Scatter Plot of 'BTC' vs 'USD'

```
plt.figure(figsize=(10, 6))
plt.scatter(df2['BTC'], df2['USD'], alpha=0.7, color='purple')
plt.title('Scatter Plot of BTC vs USD')
plt.xlabel('BTC')
plt.ylabel('USD')
plt.grid(True)
plt.show()
```

**What This Shows**:

- Reveals relationships or correlations between Bitcoin transactions and USD values.

**When to Use**:

- To detect correlations, trends, or clusters between two numerical variables.

---

## 🔥 8. Correlation Heatmaps

Heatmaps are used to visualize the correlation between numerical features in your dataset.

### Example: Correlation Heatmap

```
plt.figure(figsize=(12, 8))
corr_matrix = df2.corr()
```

```
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix Heatmap')
plt.show()
```

**What This Shows**:

- Highlights the strength and direction of correlations between different features, helping you identify features that may be redundant or highly correlated.

**When to Use**:

- When you want to understand relationships between multiple numerical variables.

---

# 🖼 9. Pair Plots

Pair plots are used to visualize relationships between multiple numerical features at once.

## Example: Pair Plot of Key Features

```
sns.pairplot(df2[['BTC', 'USD', 'Netflow_Bytes', 'Time']], hue='Clusters')
plt.title('Pair Plot of Selected Features')
plt.show()
```

**What This Shows**:

- Useful for identifying potential clusters, correlations, or outliers across several features.

**When to Use**:

- For an overall visual understanding of the interactions between multiple variables.

---

# 💡 10. Heatmap for Confusion Matrix

Confusion matrices are essential for evaluating classification models.

## Example: Confusion Matrix Heatmap for Naive Bayes Model

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

nb_pred = nb.predict(X_test)
nb_matrix = confusion_matrix(y_test, nb_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(nb_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=["Class 0", "Class 1", "Class 2"], yticklabels=["Class
0", "Class 1", "Class 2"])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix Heatmap for Naive Bayes")
plt.show()
```

**What This Shows**:

- Helps evaluate the performance of your classification model by showing where predictions went wrong.

**When to Use**:

- For understanding model performance, especially in multi-class classification.

---

# 📅 11. Time Series Analysis

If your dataset involves time-dependent data, visualizing trends over time can be valuable.

## Example: Time Series Plot for 'USD'

```
df2['Time'] = pd.to_datetime(df2['Time'])
plt.figure(figsize=(12, 6))
plt.plot(df2['Time'], df2['USD'], label='USD over Time')
plt.xlabel('Time')
plt.ylabel('USD')
plt.title('Time Series of USD Values')
plt.legend()
plt.grid(True)
plt.show()
```

**What This Shows**:

- Helps detect seasonal patterns, spikes, or anomalies in financial data.

**When to Use**:

- For datasets where the time dimension is critical for understanding trends.

---

Alright! Let's create comprehensive notes for **data visualization**, **EDA**, and **distribution analysis** that will cover **all the essential tools, customization options, and practical tips**. I'll break this into multiple sections for better clarity and understanding. Let's start with the **essential visualization tools**, then dive deeper into **customizations** and **advanced techniques**.

# Part 1: Data Visualization and Distribution Analysis

Visualizing data is a crucial part of EDA because it allows you to understand patterns, detect outliers, and identify trends in your data. Here's a breakdown of the key visualization tools you'll need.

---

## 1. Histograms

### Purpose:

Histograms are used to show the frequency distribution of a continuous variable.

### Code Example:

```
import matplotlib.pyplot as plt

# Basic histogram
plt.hist(df['Age'], bins=20, edgecolor='black')
plt.title('Distribution of Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

### Customizations:

- **Change colors**:

```
plt.hist(df['Age'], bins=20, color='skyblue', edgecolor='black')
```

- **Add grid lines**:

```
plt.grid(True)
```

- **Plot density instead of frequency**:

```
plt.hist(df['Age'], bins=20, density=True)
```

---

# 2. Box Plots

## Purpose:

Box plots show the distribution of data based on a five-number summary (minimum, first quartile, median, third quartile, and maximum) and highlight outliers.

## Code Example:

```
import seaborn as sns

# Box plot for a single variable
sns.boxplot(x=df['Income'])
plt.title('Box Plot of Income')
plt.show()
```

## Customizations:

- **Change orientation**:

```
sns.boxplot(y=df['Income'])
```

- **Group by a categorical variable**:

```
sns.boxplot(x='Category', y='Income', data=df, palette='Set3')
```

- **Remove outliers**:

```
sns.boxplot(x='Category', y='Income', data=df, showfliers=False)
```

---

# 3. Density Plots (Kernel Density Estimation)

## Purpose:

Density plots are used to estimate the probability density function of a continuous variable.

## Code Example:

```
sns.kdeplot(df['Age'], shade=True)
plt.title('Density Plot of Age')
plt.xlabel('Age')
plt.ylabel('Density')
plt.show()
```

## Customizations:

- **Change line styles**:

```
sns.kdeplot(df['Age'], linestyle='--', color='red')
```

- **Overlay multiple density plots**:

```
sns.kdeplot(df['Age'], shade=True, label='Age')
sns.kdeplot(df['Income'], shade=True, label='Income')
plt.legend()
```

---

# 4. Violin Plots

### Purpose:

Violin plots combine features of box plots and density plots, showing the distribution and density of data.

### Code Example:

```
sns.violinplot(x='Category', y='Income', data=df)
plt.title('Violin Plot of Income by Category')
plt.show()
```

### Customizations:

- **Change color palettes**:

  ```
  sns.violinplot(x='Category', y='Income', data=df, palette='coolwarm')
  ```

- **Split by a binary variable**:

  ```
  sns.violinplot(x='Category', y='Income', data=df, hue='Gender',
  split=True)
  ```

---

## 5. Pair Plots

### Purpose:

Pair plots visualize pairwise relationships between multiple numerical variables. They are useful for detecting correlations and interactions.

### Code Example:

```
sns.pairplot(df[['Age', 'Income', 'Spending_Score']], hue='Category')
plt.show()
```

### Customizations:

- **Use different markers**:

```
sns.pairplot(df, hue='Gender', markers=['o', 's'])
```

- **Add regression lines**:

```
sns.pairplot(df, kind='reg')
```

---

# 6. Heatmaps

## Purpose:

Heatmaps display the correlation matrix of numerical variables, which is helpful for identifying multicollinearity.

## Code Example:

```
corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix')
plt.show()
```

## Customizations:

- **Remove diagonal correlation values**:

```
sns.heatmap(corr_matrix, mask=np.triu(corr_matrix))
```

- **Change color schemes**:

```
sns.heatmap(corr_matrix, cmap='YlGnBu')
```

---

# 7. Scatter Plots

## Purpose:

Scatter plots show relationships between two numerical variables.

### Code Example:

```
plt.scatter(df['Age'], df['Income'], alpha=0.7, edgecolors='w',
color='purple')
plt.title('Scatter Plot of Age vs Income')
plt.xlabel('Age')
plt.ylabel('Income')
plt.grid(True)
plt.show()
```

### Customizations:

- **Add trendlines**:

  ```
  sns.regplot(x='Age', y='Income', data=df, scatter_kws={'color': 'purple'},
  line_kws={'color': 'red'})
  ```

- **Use color gradients based on a third variable**:

  ```
  plt.scatter(df['Age'], df['Income'], c=df['Spending_Score'],
  cmap='viridis')
  plt.colorbar()
  ```

---

## 8. Count Plots

### Purpose:

Count plots are used for visualizing the frequency of categorical variables.

### Code Example:

```
sns.countplot(x='Category', data=df)
plt.title('Count of Categories')
plt.show()
```

### Customizations:

- **Add hue to differentiate groups**:

```
sns.countplot(x='Category', data=df, hue='Gender', palette='Set2')
```

---

# Part 2: Advanced EDA Techniques

## 1. Analyzing Outliers

- **Box plots** and **z-score analysis** are commonly used to detect outliers.

```
from scipy import stats
z_scores = stats.zscore(df['Income'])
outliers = df[(z_scores > 3) | (z_scores < -3)]
```

## 2. Feature Engineering

- Create new features by combining existing ones or applying domain knowledge.

```
df['Income_per_Age'] = df['Income'] / df['Age']
```

## 3. Dimensionality Reduction

- **PCA (Principal Component Analysis)** for reducing feature space.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(df.drop('target', axis=1))
```

## 4. Binning Continuous Variables

- **Binning** helps convert continuous variables into categorical ones.

```
df['Age_Bins'] = pd.cut(df['Age'], bins=[0, 20, 40, 60, 80], labels=
['Young', 'Adult', 'Senior', 'Elderly'])
```

## 5. Handling Missing Values

- **Impute with mean, median, or mode**:

```
df['Income'].fillna(df['Income'].mean(), inplace=True)
```

- **Forward fill and backward fill**:

```
df['Date'].fillna(method='ffill', inplace=True)
```

---

Great! Now that we've covered data visualization and initial EDA techniques, let's dive deeper into **advanced EDA** and **feature exploration**. This section will focus on methods for gaining deeper insights into your data, including statistical tests, dealing with imbalanced data, and exploring interactions between variables.

---

# Part 3: Advanced EDA Techniques

## 1. Analyzing Feature Distributions

Before diving into model building, it's crucial to understand the distribution of your features. Let's explore some advanced tools for this.

### a) Q-Q Plots (Quantile-Quantile Plots)

**Purpose**:
Q-Q plots are used to compare the distribution of a dataset to a theoretical distribution (commonly a normal distribution). If the data follows the distribution, the points will lie along a straight line.

**Code Example**:

```
import scipy.stats as stats
import matplotlib.pyplot as plt

# Q-Q plot to check for normality
stats.probplot(df['Income'], dist="norm", plot=plt)
```

```
plt.title('Q-Q Plot of Income')
plt.show()
```

**Interpretation**:

- **Straight line**: Data is normally distributed.
- **Curved line**: Data is skewed or has heavy tails.

---

## b) Skewness and Kurtosis

**Purpose**:
Skewness measures the asymmetry of the data, while kurtosis measures the "tailedness" (extremes) of the data distribution.

**Code Example**:

```
# Skewness and Kurtosis
print("Skewness of Age:", df['Age'].skew())
print("Kurtosis of Income:", df['Income'].kurt())
```

**Interpretation**:

- **Skewness**:
  - Positive: Right-skewed (long tail on the right).
  - Negative: Left-skewed (long tail on the left).
- **Kurtosis**:
  - > 0: Heavy tails (leptokurtic).
  - < 0: Light tails (platykurtic).

---

# 2. Detecting and Handling Outliers

Outliers can heavily influence your model's performance, especially in regression and distance-based algorithms. Here are some techniques to detect and handle them.

## a) Z-Score Method

**Code Example**:

```
from scipy import stats

# Calculate Z-scores
z_scores = stats.zscore(df[['Income', 'Age', 'Netflow_Bytes']])
outliers = (abs(z_scores) > 3).any(axis=1)
df_outliers_removed = df[~outliers]

print("Number of Outliers Removed:", sum(outliers))
```

### b) IQR Method

**Code Example**:

```
Q1 = df['Income'].quantile(0.25)
Q3 = df['Income'].quantile(0.75)
IQR = Q3 - Q1

# Detect outliers
outliers = (df['Income'] < (Q1 - 1.5 * IQR)) | (df['Income'] > (Q3 + 1.5 *
IQR))
df_no_outliers = df[~outliers]

print("Number of Outliers Removed using IQR:", outliers.sum())
```

---

## 3. Feature Engineering & Interaction Analysis

Creating new features and exploring interactions between variables can significantly improve model performance.

### a) Feature Interaction

**Code Example**:

```
# Interaction feature
df['Age_Income_Interaction'] = df['Age'] * df['Income']
sns.scatterplot(x='Age_Income_Interaction', y='Spending_Score', data=df)
plt.title('Interaction between Age and Income')
plt.show()
```

### b) Polynomial Features

**Code Example**:

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(df[['Age', 'Income']])
poly_df = pd.DataFrame(poly_features,
columns=poly.get_feature_names_out(['Age', 'Income']))
df = pd.concat([df, poly_df], axis=1)
```

---

# 4. Handling Imbalanced Datasets

Imbalanced datasets can lead to biased models, especially in classification problems. Let's look at techniques to address this issue.

## a) Visualizing Class Imbalance

**Code Example**:

```
sns.countplot(x='Target', data=df)
plt.title('Class Distribution')
plt.show()
```

## b) Resampling Techniques

- **Oversampling**: Increase the number of samples in the minority class.

```
from imblearn.over_sampling import SMOTE

smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X, y)
```

- **Undersampling**: Reduce the number of samples in the majority class.

```
from imblearn.under_sampling import RandomUnderSampler

undersample = RandomUnderSampler()
X_resampled, y_resampled = undersample.fit_resample(X, y)
```

# 5. Statistical Tests for Feature Selection

## a) Chi-Square Test (For Categorical Features)

**Purpose**:
Used to check if there is an association between two categorical variables.

**Code Example**:

```
from sklearn.feature_selection import chi2
from sklearn.feature_selection import SelectKBest

chi2_selector = SelectKBest(score_func=chi2, k=5)
X_new = chi2_selector.fit_transform(X, y)
print("Top 5 features:", chi2_selector.get_feature_names_out())
```

## b) ANOVA F-Test (For Continuous Features)

**Purpose**:
Used to identify features with the greatest variance.

**Code Example**:

```
from sklearn.feature_selection import f_classif

f_values, p_values = f_classif(X, y)
print("ANOVA F-values:", f_values)
print("P-values:", p_values)
```

---

# 6. Correlation Analysis for Feature Selection

## a) Visualizing Correlation Matrix

**Code Example**:

```
corr_matrix = df.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
```

```
plt.title('Correlation Matrix')
plt.show()
```

### b) Feature Selection Based on Correlation

- **Remove highly correlated features** to reduce multicollinearity.

```
correlated_features = set()
for i in range(len(corr_matrix.columns)):
    for j in range(i):
        if abs(corr_matrix.iloc[i, j]) > 0.9:
            correlated_features.add(corr_matrix.columns[i])

df.drop(columns=correlated_features, inplace=True)
```

---

# Part 4: Advanced EDA Customizations

## 1. Customizing Plot Styles

```
# Set global style
plt.style.use('seaborn-darkgrid')

# Change font size and background color
plt.rcParams.update({'font.size': 12, 'figure.facecolor': 'white'})

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

## 2. Saving High-Quality Plots

- **Save plots in different formats** (PNG, PDF, etc.) for reports.

```
plt.savefig('correlation_matrix.png', dpi=300, bbox_inches='tight')
```

## 3. Interactive Visualizations with Plotly

```
import plotly.express as px
```

```
fig = px.scatter(df, x='Age', y='Income', color='Category', hover_data=
['Spending_Score'])
fig.update_layout(title='Interactive Scatter Plot')
fig.show()
```

---

# Conclusion

These comprehensive notes cover both basic and advanced tools for EDA, feature engineering, and distribution analysis. By mastering these techniques, you'll be able to conduct thorough EDA, uncover hidden patterns, and make informed decisions for feature selection.

Next, we can explore **model training and evaluation techniques**, **hyperparameter tuning**, or **explainable AI (XAI)** methods if you're interested!

Excellent question! We've already covered a broad range of visualization techniques, but there are **additional tools and methods** that can give deeper insights into your data, especially for complex datasets. Let's expand further with some **advanced visualization techniques**, **custom visualizations**, and **specialized plots** that are often used in data science to enhance analysis.

We'll also cover some **interactive and advanced libraries** for dynamic visualizations that can make your analysis more intuitive.

---

# Part 5: Additional Visualization Techniques

## 1. Distribution Visualizations Beyond Basic Histograms

### a) Violin Plot

**Purpose**: Combines a box plot and a kernel density plot to show the distribution of the data across several categories. Useful for understanding the distribution and density.

**Code Example**:

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
sns.violinplot(x='Category', y='Income', data=df)
```

```
plt.title('Violin Plot of Income by Category')
plt.show()
```

**Interpretation**:

- Thicker sections show where data points are more concentrated.
- Useful for comparing distributions between different categories.

---

## b) Pair Plot

**Purpose**: Displays pairwise relationships in a dataset. Useful for exploring interactions between features, especially in higher dimensions.

**Code Example**:

```
sns.pairplot(df, hue='Target', diag_kind='kde')
plt.title('Pair Plot of Features')
plt.show()
```

**Customizations**:

- `hue`: Color by categorical feature.
- `diag_kind`: Choose between histogram or KDE for diagonal plots.

---

# 2. Time Series Visualizations

## a) Line Plot with Rolling Average

**Purpose**: Useful for identifying trends and smoothing fluctuations in time series data.

**Code Example**:

```
df['Rolling_Avg'] = df['Sales'].rolling(window=7).mean()
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['Sales'], label='Original')
plt.plot(df['Date'], df['Rolling_Avg'], label='7-Day Rolling Average',
color='red')
plt.legend()
```

```
plt.title('Sales Over Time with Rolling Average')
plt.show()
```

---

## b) Autocorrelation Plot

**Purpose**: Checks if there is a correlation between the current value and its previous values in time series data.

**Code Example**:

```
from pandas.plotting import autocorrelation_plot

autocorrelation_plot(df['Sales'])
plt.title('Autocorrelation Plot')
plt.show()
```

---

# 3. Categorical Data Visualizations

## a) Heatmap with Annotations

**Purpose**: Displays correlation between categorical variables and their frequency.

**Code Example**:

```
cross_tab = pd.crosstab(df['Gender'], df['Purchased'])
sns.heatmap(cross_tab, annot=True, cmap='Blues')
plt.title('Heatmap of Gender vs Purchased')
plt.show()
```

**Customization**: Use different color maps ( `cmap` ) like `'coolwarm'` , `'YlGnBu'` , or `'magma'` .

---

## b) Mosaic Plot

**Purpose**: Shows the relationship between two or more categorical variables. Useful for understanding distribution and proportions.

**Code Example**:

```
from statsmodels.graphics.mosaicplot import mosaic

mosaic(df, ['Gender', 'Purchased'])
plt.title('Mosaic Plot of Gender vs Purchased')
plt.show()
```

---

# 4. Multivariate Visualizations

## a) Radar Chart (Spider Plot)

**Purpose**: Compares multiple variables for several categories on a radial plot. Useful for comparing performance metrics or profile analysis.

**Code Example**:

```
from math import pi

categories = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
values = [0.85, 0.75, 0.80, 0.78]
angles = [n / float(len(categories)) * 2 * pi for n in range(len(categories))]
angles += angles[:1]  # Complete the loop

plt.figure(figsize=(8, 6))
ax = plt.subplot(111, polar=True)
plt.xticks(angles[:-1], categories)
ax.plot(angles, values + values[:1])
ax.fill(angles, values + values[:1], alpha=0.3)
plt.title('Radar Chart for Model Performance')
plt.show()
```

---

## b) Parallel Coordinates Plot

**Purpose**: Used to visualize multi-dimensional data by plotting all features for each sample on parallel axes.

**Code Example**:

```
from pandas.plotting import parallel_coordinates

parallel_coordinates(df, 'Target', color=['#FF6347', '#4682B4'])
plt.title('Parallel Coordinates Plot')
plt.show()
```

---

# 5. Advanced Dimensionality Reduction Visualizations

## a) t-SNE (t-Distributed Stochastic Neighbor Embedding)

**Purpose**: Reduces high-dimensional data into two or three dimensions for visualization while preserving cluster structures.

**Code Example**:

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(X)
plt.figure(figsize=(10, 8))
plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y, cmap='viridis')
plt.title('t-SNE Plot')
plt.show()
```

---

## b) UMAP (Uniform Manifold Approximation and Projection)

**Purpose**: Similar to t-SNE but generally faster and better at preserving global structures in high-dimensional data.

**Code Example**:

```
import umap

umap_model = umap.UMAP(n_neighbors=15, n_components=2, random_state=42)
embedding = umap_model.fit_transform(X)
plt.scatter(embedding[:, 0], embedding[:, 1], c=y, cmap='plasma')
plt.title('UMAP Visualization')
plt.show()
```

# 6. Interactive Visualization Libraries

## a) Plotly

- **Purpose**: Create interactive plots that can be zoomed, hovered, and dynamically updated.

**Code Example**:

```
import plotly.express as px

fig = px.scatter(df, x='Age', y='Income', color='Target', hover_data=
['Spending_Score'])
fig.update_layout(title='Interactive Scatter Plot')
fig.show()
```

---

## b) Bokeh

- **Purpose**: Useful for creating interactive dashboards and linking multiple plots.

**Code Example**:

```
from bokeh.plotting import figure, show
from bokeh.io import output_notebook

output_notebook()
p = figure(title='Bokeh Line Plot', x_axis_label='Date', y_axis_label='Sales')
p.line(df['Date'], df['Sales'], line_width=2)
show(p)
```

---

# 7. Customizing Visualizations

- **Adding Titles, Labels, and Legends**:

```
plt.title('Plot Title')
plt.xlabel('X-axis Label')
```

```
plt.ylabel('Y-axis Label')
plt.legend(['Label 1', 'Label 2'])
```

- **Changing Color Palettes**:

```
sns.set_palette('Set2')
sns.barplot(x='Category', y='Values', data=df)
```

- **Styling with Matplotlib**:

```
plt.style.use('ggplot')
plt.figure(figsize=(10, 6))
```

- **Saving Plots**:

```
plt.savefig('plot.png', dpi=300, bbox_inches='tight')
```

---

Got it! Let's include **adjusting text sizes** for various elements like axis labels, titles, legends, and annotations. I'll update the examples to show how to control text sizes for each component in your plots.

---

# 1. Plotting Distributions with Normal Distribution Line and Text Size Adjustments

## 1.1 Plotting with Seaborn and Customizing Text Sizes

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm

# Generate sample data
data = np.random.normal(loc=50, scale=10, size=1000)

# Set plot style
plt.style.use('seaborn-darkgrid')
```

```
# Plot histogram with density estimation
sns.histplot(data, kde=True, stat='density', color='skyblue')

# Fit a normal distribution to the data
mean, std_dev = norm.fit(data)
x = np.linspace(min(data), max(data), 100)
y = norm.pdf(x, mean, std_dev)
plt.plot(x, y, color='red', linestyle='dashed')

# Adding labels with custom text sizes
plt.title('Histogram with Normal Distribution Curve', fontsize=18)
plt.xlabel('Data Values', fontsize=14)
plt.ylabel('Density', fontsize=14)

# Adjust tick label sizes
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Add grid
plt.grid(True)
plt.show()
```

### Explanation:

- `fontsize` is used to adjust the text size for the title, labels, and ticks.
- You can customize the plot appearance using `plt.style.use()`.

---

## 2. Adding Annotations and Lines with Custom Text Sizes

### 2.1 Adding Vertical Line, Text Annotation, and Custom Font Sizes

```
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate sample data
data = np.random.normal(0, 1, 1000)

# Plot histogram
```

```
plt.figure(figsize=(10, 6))
plt.hist(data, bins=30, density=True, color='lightblue', edgecolor='black')

# Fit a normal distribution curve
mean, std_dev = norm.fit(data)
x = np.linspace(min(data), max(data), 100)
y = norm.pdf(x, mean, std_dev)
plt.plot(x, y, 'r-', linewidth=2)

# Add a vertical line at the mean
plt.axvline(mean, color='green', linestyle='--', linewidth=1.5)

# Adding text annotation
plt.text(mean + 0.2, 0.35, f'Mean = {mean:.2f}', fontsize=14, color='green')

# Annotate with an arrow
plt.annotate('Peak of the distribution',
             xy=(mean, 0.4),
             xytext=(mean + 0.5, 0.5),
             arrowprops=dict(facecolor='black', arrowstyle='->'),
             fontsize=12)

# Customize labels and title
plt.title('Distribution Plot with Annotations', fontsize=20)
plt.xlabel('Data Value', fontsize=16)
plt.ylabel('Density', fontsize=16)

# Adjust tick label sizes
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

plt.grid(True)
plt.show()
```

## Explanation:

- **Annotations**: Added with `plt.annotate()` using custom `fontsize`.
- **Text and Lines**: Adjusted with `plt.text()` and `plt.axvline()`.

---

# 3. Comprehensive Example: Customizing All Text Elements

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm

# Generate random data
data = np.random.normal(0, 1, 1000)

# Set figure size and style
plt.figure(figsize=(12, 7))
plt.style.use('ggplot')

# Histogram with density plot
sns.histplot(data, kde=True, stat='density', color='skyblue', alpha=0.6)

# Fit a normal distribution
mean, std_dev = norm.fit(data)
x = np.linspace(min(data), max(data), 100)
y = norm.pdf(x, mean, std_dev)
plt.plot(x, y, color='red', linestyle='-', linewidth=2)

# Vertical line for the mean
plt.axvline(mean, color='green', linestyle='--', linewidth=1.5)
plt.text(mean + 0.05, 0.3, f'Mean: {mean:.2f}', fontsize=14, color='green')

# Adding a title and axis labels with custom font sizes
plt.title('Histogram with Normal Distribution Line', fontsize=22,
fontweight='bold')
plt.xlabel('Values', fontsize=16, fontweight='bold')
plt.ylabel('Density', fontsize=16, fontweight='bold')

# Adding a legend with custom font size
plt.legend(['Normal Distribution', 'Mean'], fontsize=12)

# Customizing ticks
plt.xticks(fontsize=12, rotation=45)
plt.yticks(fontsize=12)

# Add grid with custom style
plt.grid(True, linestyle='--', linewidth=0.5)

# Show plot
plt.show()
```

## Explanation:

- **Legend**: Adjusted with `plt.legend()` and custom `fontsize`.
- **Ticks**: Adjusted with `plt.xticks()` and `plt.yticks()`.
- **Title and Labels**: Adjusted with `fontsize` and `fontweight`.

---

# 4. Saving the Plot with High Resolution

## 4.1 Saving with Adjustments

```
plt.savefig('distribution_plot.png', dpi=300, bbox_inches='tight',
transparent=True)
```

- `dpi=300`: High resolution for clear images.
- `bbox_inches='tight'`: Trims any extra whitespace.
- `transparent=True`: Saves the plot with a transparent background.

---

# 5. Quick Reference for Font Size Adjustments

- **Title**: `plt.title('Title', fontsize=20)`
- **X-axis label**: `plt.xlabel('X Label', fontsize=14)`
- **Y-axis label**: `plt.ylabel('Y Label', fontsize=14)`
- **Tick labels**: `plt.xticks(fontsize=12)` / `plt.yticks(fontsize=12)`
- **Legend**: `plt.legend(fontsize=12)`
- **Text annotation**: `plt.text(x, y, 'Text', fontsize=12)`
- **Annotations with arrow**: `plt.annotate('Text', xy=(x, y), xytext=(x+1, y+1), arrowprops=dict(facecolor='black'))`

---

These notes cover how to plot distributions, customize text sizes, add annotations, and adjust various elements to make your visualizations more readable and professional. Let me know if you need further customization or specific examples!