PROFESSIONAL ELECTIVE COURSES: VERTICALS
VERTICAL 1: DATA SCIENCE
CCS346 EXPLORATORY DATA ANALYSIS
UNIT V - MULTIVARIATE AND TIME SERIES ANALYSIS

Introducing a Third Variable – Causal Explanations – Three-Variable Contingency Tables and Beyond – Fundamentals of TSA – Characteristics of time series data – Data Cleaning – Time-based indexing – Visualizing – Grouping – Resampling.

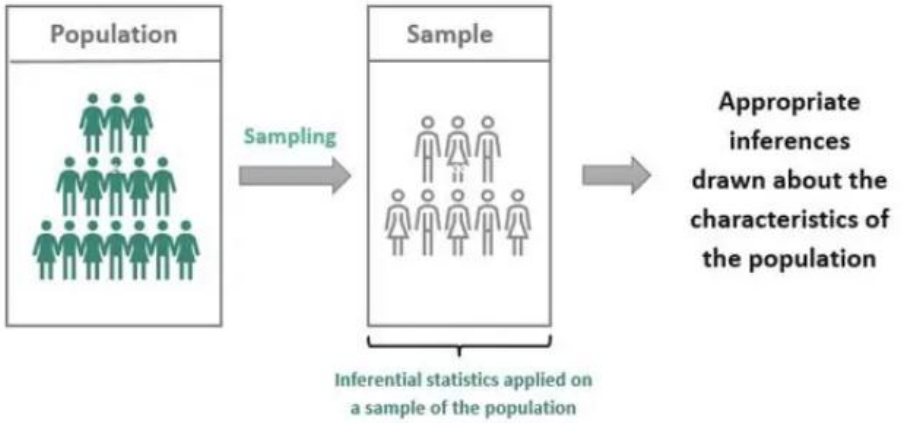
Inferential Statistics

Inferential statistics is a branch of statistics that makes the use of various analytical tools to draw inferences about the population data from sample data. The purpose of descriptive and inferential statistics is to analyze different types of data using different tools. Descriptive statistics helps to describe and organize known data using charts, bar graphs, etc., while inferential statistics aims at making inferences and generalizations about the population data.

Descriptive Statistics		Inferential Statistics	
Measures of Central Tendency	Measures of Dispersion	Hypothesis Testing	Regression Analysis
Mean Median Mode	Range Standard Deviation Variance Absolute Deviation	Z test F test T test	Linear Regression

Descriptive statistics allow you to describe a data set, while inferential statistics allow you to make inferences based on a data set. The samples chosen in inferential statistics need to be representative of the entire population.

Descriptive statistics	Inferential statistics
Collecting, summarizing and Describing Data	Drawing conclusions and/or making decisions from the sample of population



There are two main types of inferential statistics - hypothesis testing and regression analysis.

- Hypothesis Testing - This technique involves the use of hypothesis tests such as the z test, f test, t test, etc. to make inferences about the population data. It requires setting up the null hypothesis, alternative hypothesis, and testing the decision criteria.
- Regression Analysis - Such a technique is used to check the relationship between dependent and independent variables. The most commonly used type of regression is linear regression.

Inferential Statistics	
Hypothesis Testing	Regression Analysis
Z test	Linear Regression
F test	Nominal Regression
T test	Logistic Regression
ANOVA Test	Ordinal Regression
Wilcoxon Signed Rank Test	
Mann-Whitney U Test	

<u>Brief about all tests</u>

Z-test

- Sample size is greater than or equal to 30 and the data set follows a normal distribution.
- The population variance is known to the researcher.
    - $Null\ hypothesis: H0 :\ \mu = \mu 0$
    - $Alternate\ hypothesis: H1: \mu > \mu 0$

$$\text{Z Test} = \frac{\bar{x} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

T-test

- Sample size is less than 30 and the data set follows a t-distribution.
- The population variance is not known to the researcher.
  - $Null\ Hypothesis$: $H0: \mu = \mu 0$
  - $Alternate\ Hypothesis$: $H1: \mu > \mu 0$

$$t = \frac{\bar{x} - \mu}{\frac{s}{\sqrt{n}}}$$

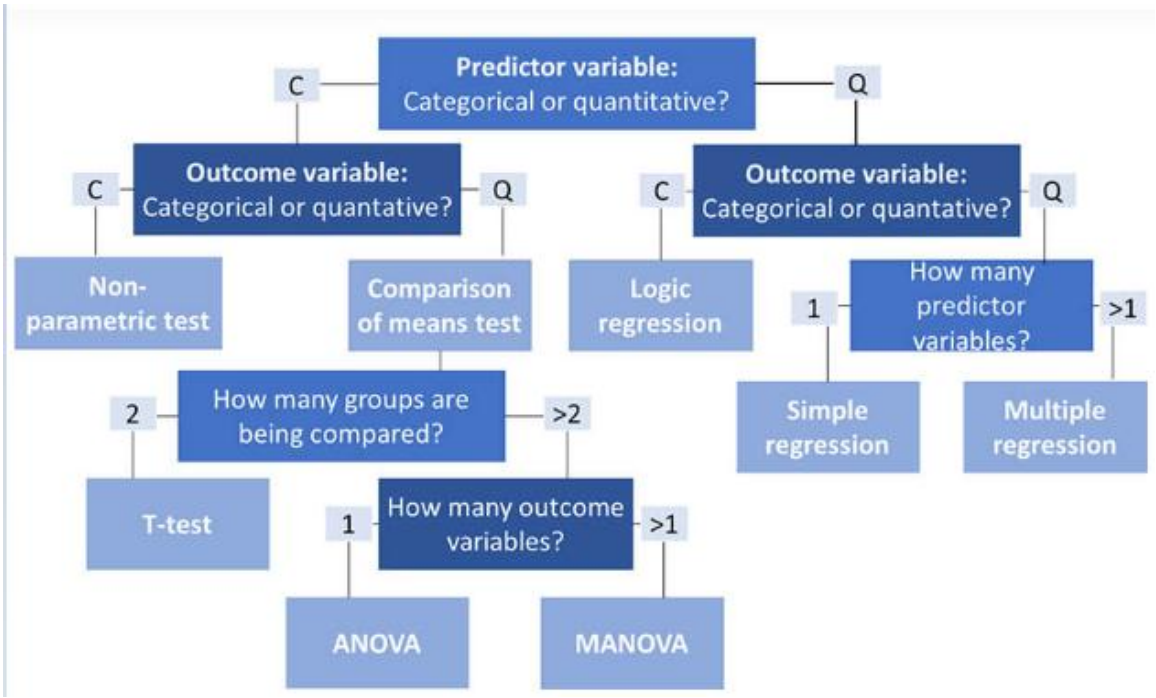
F-test

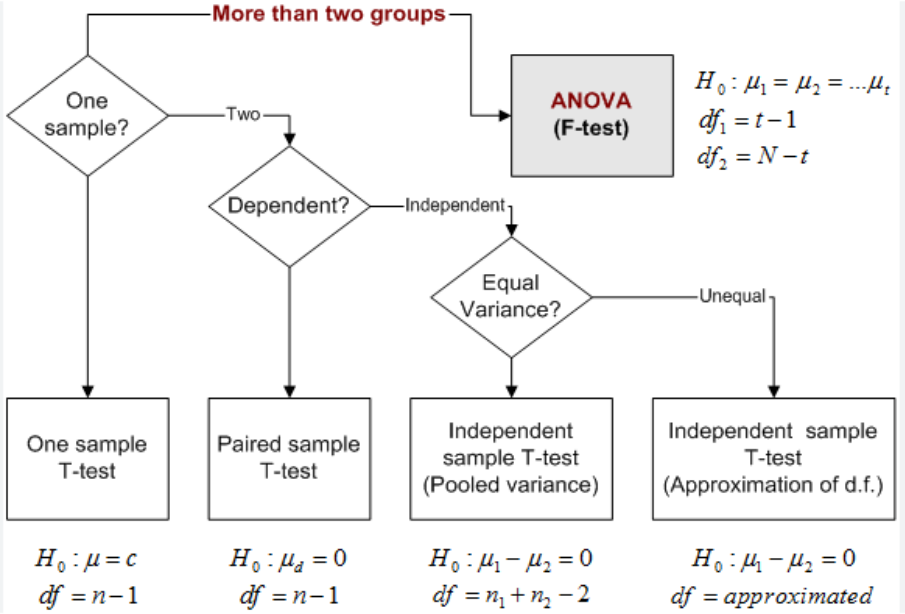
- Checks whether a difference between the variances of two samples or populations exists or not.

$$\text{F Value} = \frac{\text{Variance of 1st Data Set}}{\text{Variance of 2nd Data Set}}$$

Choosing a statistical test:

The following flowchart allows you to choose the correct statistical test for your research easily.





Multivariate Analysis:

Multivariate analysis refers to statistical techniques used to analyze and understand relationships between multiple variables simultaneously. It involves exploring patterns, dependencies, and associations among variables in a dataset. Some commonly used multivariate analysis techniques include:

- Multivariate Regression Analysis: Extends simple linear regression to analyze the relationship between multiple independent variables and a dependent variable.
- Principal Component Analysis (PCA): Reduces the dimensionality of a dataset by transforming variables into a smaller set of uncorrelated variables called principal components.
- Factor Analysis: Examines the underlying factors or latent variables that explain the correlations among a set of observed variables.
- Cluster Analysis: Identifies groups or clusters of similar observations based on the similarity of their attributes.
- Discriminant Analysis: Differentiates between two or more predefined groups based on a set of predictor variables.
- Canonical Correlation Analysis: Analyzes the relationship between two sets of variables to identify the underlying dimensions that are shared between them.

<u>Simple Example of Multivariate Analysis</u>

Let's consider a simple example of multivariate analysis using a dataset that includes three variables: "age" (continuous), "income" (continuous), and "education level" (categorical: high school, bachelor's degree, master's degree).

Objective: Determine the relationship between age, income, and education level.

Approach:

Descriptive Analysis:

- Calculate descriptive statistics for each variable, including measures like mean, median, standard deviation, and frequency distributions.
- Examine the distributions of age and income using histograms or boxplots to identify any outliers or unusual patterns.
- Create cross-tabulations or contingency tables to explore the distribution of education level across different age groups or income brackets.

Correlation Analysis:

- Calculate the correlation coefficients (e.g., Pearson correlation, Spearman correlation) between age, income, and education level.
- Interpret the correlation coefficients to determine the strength and direction of the relationships between variables.
- Visualize the relationships using a correlation matrix or a heatmap to identify any significant associations.

Regression Analysis:

- Perform multivariate regression analysis to assess the impact of age and education level on income.
- Set income as the dependent variable and age and education level as independent variables.
- Interpret the regression coefficients to understand how each independent variable influences the dependent variable.
- Assess the overall model fit and statistical significance of the regression model.

Multivariate Visualization:

- Create scatter plots or bubble plots to visualize the relationship between age, income, and education level.
- Use different colors or symbols to represent different education levels and examine if there are distinct patterns or trends.

Further Analysis:

- Consider additional multivariate techniques such as factor analysis or cluster analysis to explore underlying dimensions or groups within the data.
- Conduct subgroup analyses or interaction analyses to investigate if the relationships differ across different demographic groups or educational backgrounds.

Causal explanations

Causal explanations aim to understand the cause-and-effect relationships between variables and explain why certain outcomes occur. They involve identifying the factors or

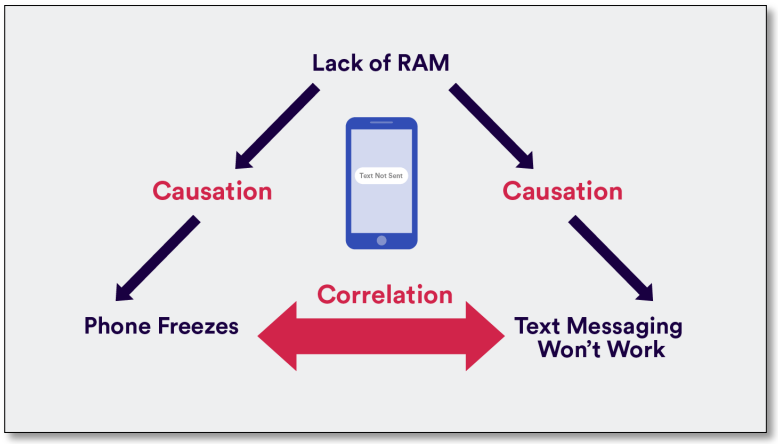
conditions that influence a particular outcome and determining the mechanisms through which they operate.

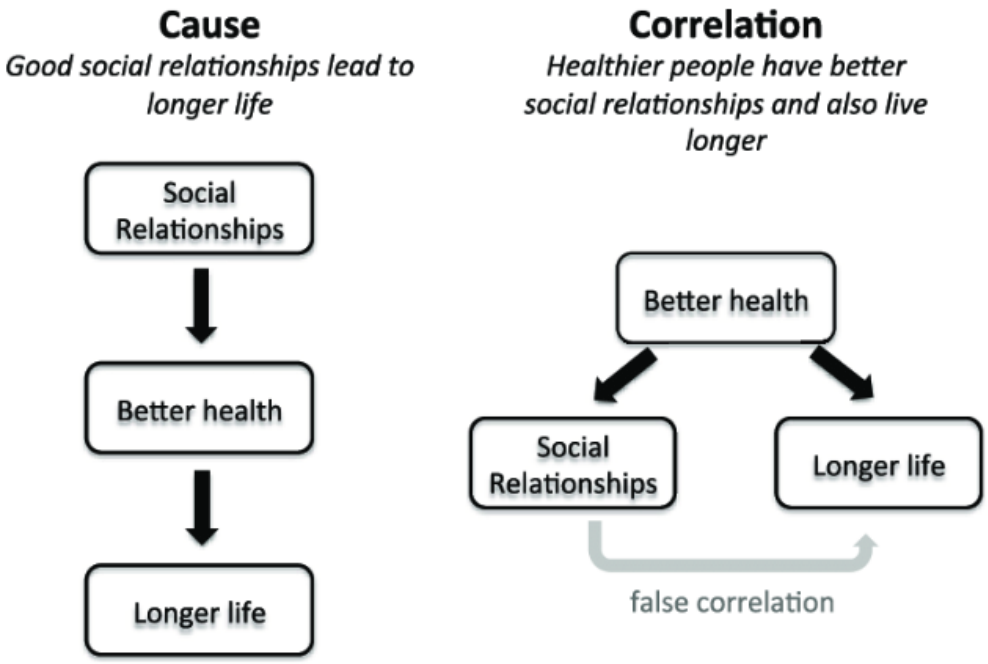
Causal explanations are important in various fields, including social sciences, economics, psychology, and epidemiology, among others. They help researchers understand the fundamental drivers of phenomena and develop interventions or policies to bring about desired outcomes.

Some key aspects and approaches to consider when seeking causal explanations:

Association vs. Causation:

It's crucial to differentiate between mere associations or correlations between variables and actual causal relationships. Correlation does not imply causation, and establishing causality requires rigorous evidence, such as experimental designs or well-designed observational studies that account for potential confounding factors.





Establishing Causality:
Several criteria need to be considered when establishing causality, such as temporal precedence (the cause precedes the effect in time), covariation (the cause and effect vary together), and ruling out alternative explanations.

Simple Scenario:
We want to investigate whether exercise has a causal effect on weight loss. We hypothesize that regular exercise leads to a reduction in weight.

Explanation:
To establish a causal explanation, we would need to conduct a study that meets the criteria for establishing causality, such as a randomized controlled trial (RCT). In this hypothetical RCT, we randomly assign participants to two groups:

- Experimental Group: Participants in this group are instructed to engage in a structured exercise program, such as 30 minutes of moderate-intensity aerobic exercise five times a week.
- Control Group: Participants in this group do not receive any specific exercise instructions and maintain their usual daily activities.

The study is conducted over a period of three months, during which the weight of each participant is measured at the beginning and end of the study. The data collected are as follows:

Experimental Group:

- Participant 1: Weight at the beginning = 80 kg, Weight at the end = 75 kg
- Participant 2: Weight at the beginning = 75 kg, Weight at the end = 72 kg
- Participant 3: Weight at the beginning = 85 kg, Weight at the end = 80 kg

Control Group:

- Participant 4: Weight at the beginning = 82 kg, Weight at the end = 81 kg
- Participant 5: Weight at the beginning = 78 kg, Weight at the end = 78 kg
- Participant 6: Weight at the beginning = 79 kg, Weight at the end = 80 kg

Analysis:

We compare the average weight loss between the experimental and control groups. The results show that the experimental group had an average weight loss of 4 kg, while the control group had an average weight loss of only 1 kg. The difference in average weight loss between the groups suggests that regular exercise has a causal effect on weight loss.

Additionally, we can use statistical tests, such as t-tests or analysis of variance (ANOVA), to determine if the observed difference in weight loss between the groups is statistically significant. If the p-value is below a predetermined significance level (e.g., $p < 0.05$), we can conclude that the difference is unlikely due to chance alone and provides further evidence for a causal relationship.

Three-variable contingency tables

When exploring causal explanations, particularly in the context of three variables, three-variable contingency tables can be utilized. These tables provide a way to examine the relationship between three categorical variables and investigate potential causal explanations by introducing a third variable.

A three-variable contingency table consists of rows and columns representing the categories of three variables, and the cells of the table contain frequency counts or proportions. It allows for the analysis of the joint distribution of three variables and helps identify any associations or dependencies between them.

Example

Let's consider the variables "Gender" (Male/Female), "Education Level" (High school/College/Graduate), and "Income Level" (Low/Medium/High). We want to explore if there is an association between gender, education level, and income level.

A three-variable contingency table for this example might look like:

Education Level	**Income Level**		
	Low	**Medium**	**High**
High School	20	40	30

College	30	50	40
Graduate	10	20	30

From this contingency table, we can analyze the relationship between these variables. For example:

- Conditional Relationships: We can examine the relationship between gender and income level, conditional on education level. This can be done by comparing the income level distribution for males and females within each education level category.
- Marginal Relationships: We can examine the relationship between gender and education level, and between education level and income level separately by looking at the marginal distributions of the variables.
- Assessing Dependency: We can perform statistical tests, such as the chi-square test, to determine if there is a statistically significant association between the variables. This helps assess the dependency and provides insights into potential causal explanations.

By analyzing the three-variable contingency table, we can gain a deeper understanding of the relationships between the variables and explore potential causal explanations by considering the influence of the third variable.

Crosstabs is just another name for contingency tables, which summarize the relationship between different categorical variables. Crosstabs in SPSS can help you visualize the proportion of cases in subgroups.

- To describe a single categorical variable, we use frequency tables.
- To describe the relationship between two categorical variables, we use a special type of table called a cross-tabulation (or "crosstab")
    - Categories of one variable determine the rows of the table
    - Categories of the other variable determine the columns
    - The cells of the table contain the number of times that a particular combination of categories occurred.

A "square" crosstab is one in which the row and column variables have the same number of categories. Tables of dimensions 2x2, 3x3, 4x4, etc. are all square crosstabs.

Example 1

Count

		Do you drink alcohol?		
		No	Yes	Total
Gender	Male	13	33	46
	Female	13	32	45
Total		26	65	91

Row variable: Gender (2 categories: male, female)
Column variable: Alcohol (2 categories: no, yes)
Table dimension: 2x2

Example 2

Class Rank * Gender Crosstabulation

Count

		Gender		
		Male	Female	Total
Class Rank	Freshman	14	9	23
	Sophomore	15	13	28
	Junior	9	11	20
	Senior	12	14	26
Total		50	47	97

Row variable: Class Rank (4 categories: freshman, sophomore, junior, senior)
Column variable: Gender (2 categories: male, female)
Table dimension: 4x2

Example 3

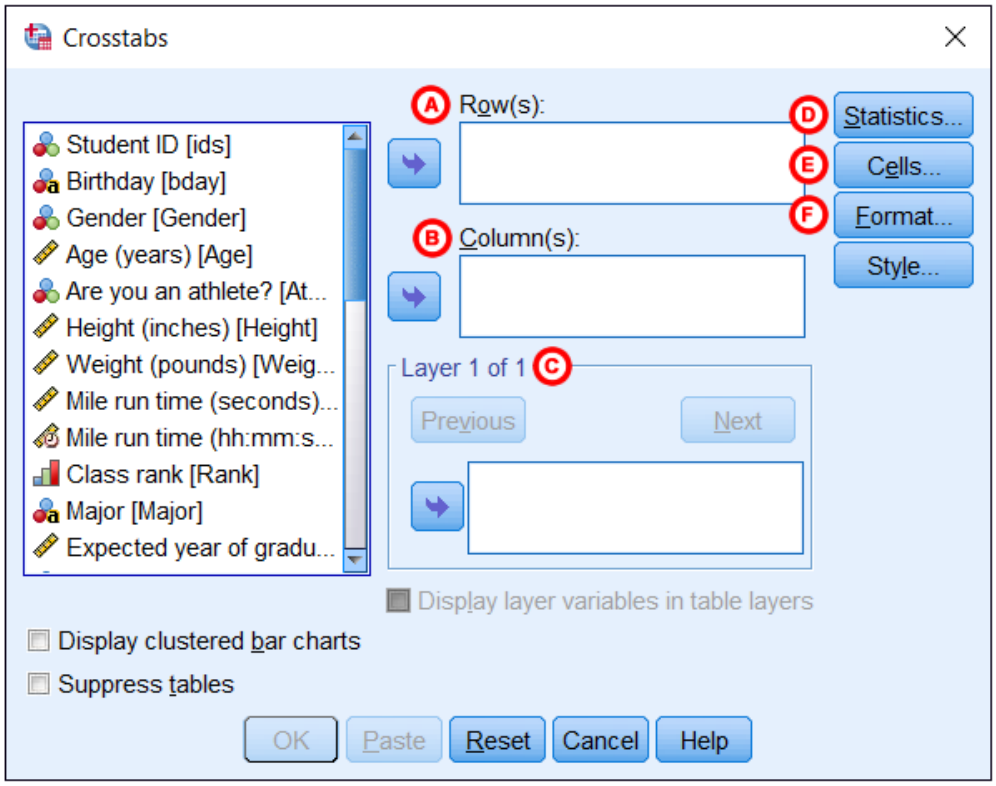
Count

		Do you smoke cigarettes?			
		Never smoked	Past smoker	Current smoker	Total
Gender	Male	32	1	14	47
	Female	25	3	16	44
Total		57	4	30	91

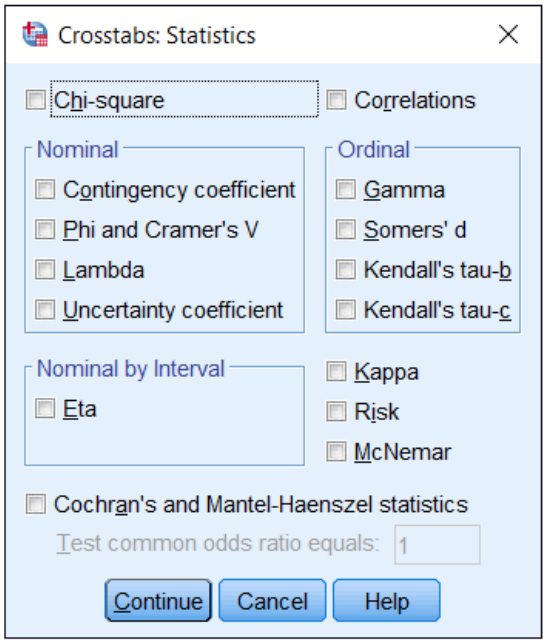
Row variable: Gender (2 categories: male, female)
Column variable: Smoking (3 categories: never smoked, past smoker, current smoker)
Table dimension: 2x3

Crosstabs with Layer Variable (Third categorical variable)

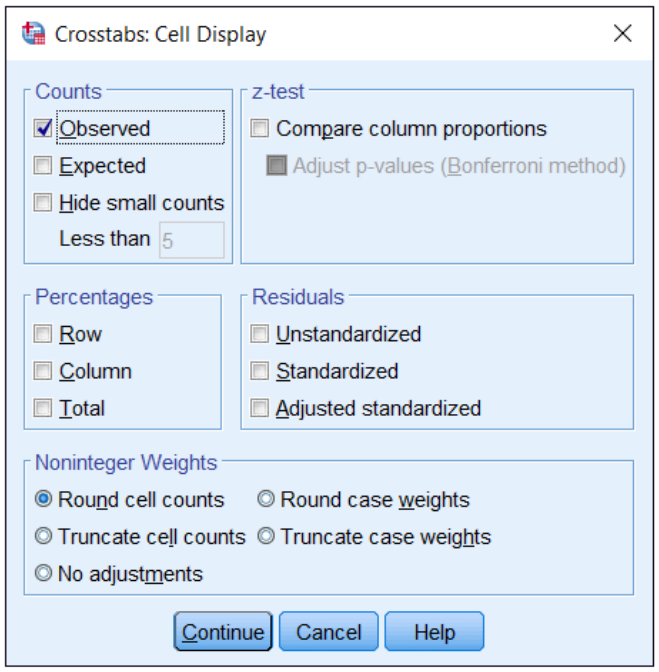
To create a crosstab, click Analyze > Descriptive Statistics > Crosstabs.



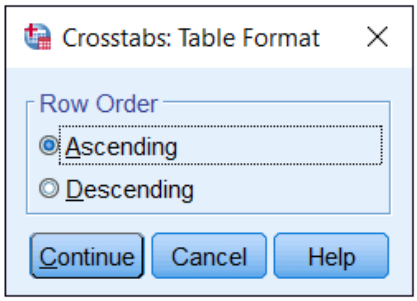
- **A** → Row(s): One or more variables to use in the rows of the crosstab(s). You must enter at least one Row variable.
- **B** →Column(s): One or more variables to use in the columns of the crosstab(s). You must enter at least one Column variable.
- **C** → Layer: An optional "stratification" variable. When a layer variable is specified, the crosstab between the Row and Column variable(s) will be created at each level of the layer variable. You can have multiple layers of variables by specifying the first layer variable and then clicking Next to specify the second layer variable.
- **D** → Statistics: Opens the Crosstabs: Statistics window, which contains fifteen different inferential statistics for comparing categorical variables.



- **E** → Cells: Opens the Crosstabs: Cell Display window, which controls which output is displayed in each cell of the crosstab.



- **F** → Format: Opens the Crosstabs: Table Format window, which specifies how the rows of the table are sorted.



Working with third categorical variable

Now we work with three categorical variables: RankUpperUnder, LiveOnCampus, and State_Residency of the student's database.

Description

Create a crosstab of RankUpperUnder by LiveOnCampus, with variable State_Residency acting as a strata, or layer variable.

Running the Procedure

- Using the Crosstabs Dialog Window
- Open the Crosstabs dialog (Analyze > Descriptive Statistics > Crosstabs).
- Select RankUpperUnder as the row variable, and LiveOnCampus as the column variable.
- Select State_Residency as the layer variable.
- You may want to go back to the Cells options and turn off the row, column, and total percentages if you have just run the previous example.
- Click OK.

Syntax

```
CROSSTABS
  /TABLES=RankUpperUnder BY LiveOnCampus BY State_Residency
  /FORMAT=AVALUE TABLES
  /CELLS=COUNT
  /COUNT ROUND CELL.
```

Output

Again, the Crosstabs output includes the boxes Case Processing Summary and the crosstabulation itself.

Case Processing Summary

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
Class Rank * Do you live on campus? * State of residence	367	84.4%	68	15.6%	435	100.0%

Notice that after including the layer variable State Residency, the number of valid cases we have to work with has dropped from 388 to 367. This is because the crosstab requires nonmissing values for all three variables: row, column, and layer.

Class Rank * Do you live on campus? * State of residence Crosstabulation

Count

State of residence			Do you live on campus?		Total
			Off-campus	On-campus	
In state	Class Rank	Underclassman	58	110	168
		Upperclassman	108	7	115
	Total		166	117	283
Out of state	Class Rank	Underclassman	13	30	43
		Upperclassman	39	2	41
	Total		52	32	84
Total	Class Rank	Underclassman	71	140	211
		Upperclassman	147	9	156
	Total		218	149	367

l

The layered crosstab shows the individual Rank by Campus tables within each level of State Residency. Some observations we can draw from this table include:

- A slightly higher proportion of out-of-state underclassmen live on campus (30/43) than do in-state underclassmen (110/168).
- There were about equal numbers of out-of-state upper and underclassmen; for in-state students, the underclassmen outnumbered the upperclassmen.
- Of the nine upperclassmen living on-campus, only two were from out of state.

Time Series Analysis (TSA)

Time Series Analysis (TSA) is a statistical technique used to analyze and understand data that is collected over time. It involves studying the patterns, trends, and characteristics of the data to make forecasts, identify underlying factors, and make informed decisions. Here are the key fundamentals of TSA:

Time Series Data:

- Time series data is a sequence of observations collected at regular time intervals. It can be in the form of numerical values, counts, percentages, or categorical data.
- The data points are ordered chronologically, and each observation is associated with a specific time stamp.

Temporal Dependencies:

- Time series data often exhibits temporal dependencies, where each observation is influenced by previous observations.
- Understanding these dependencies is crucial for analyzing and forecasting time series data accurately.

Components of Time Series:

Time series data can be decomposed into several components:

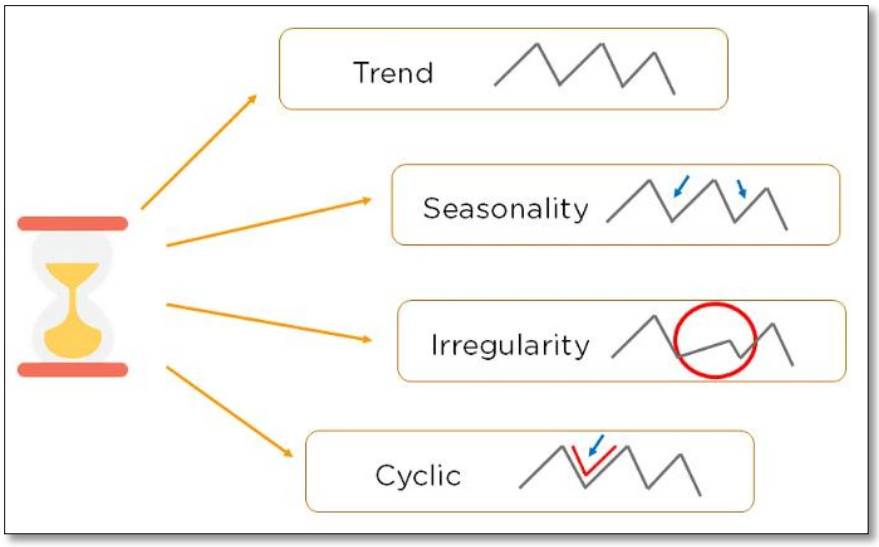
- Trend: The long-term movement or direction of the data.
    - Example: Monthly Average Home Prices in a City
    - The average home prices have been steadily increasing over the past few years, indicating a positive trend.

- Seasonality: The recurring patterns or cycles that occur at fixed time intervals.
    - Example: Monthly Sales of Ice Cream
    - Sales of ice cream are higher during the summer months compared to the rest of the year, showing a seasonal pattern.

- Cyclical Variation: Longer-term patterns that are not necessarily fixed.
    - Example: Quarterly GDP Growth Rate
    - The GDP growth rate exhibits alternating periods of expansion and contraction, representing broader cyclical patterns influenced by economic factors.

- Residuals: The random fluctuations or noise in the data.
    - Example: Daily Stock Market Returns
    - Sudden spikes or drops in stock market returns that cannot be attributed to any specific trend or seasonality are considered irregular components.

- Stationarity: It refers to the statistical properties of a time series remaining constant over time.
    - A stationary time series exhibits a constant mean, variance, and autocovariance structure.
    - Stationarity is important because many time series models assume stationarity to make reliable forecasts.



Time Series Visualization:

- Visualizing time series data helps in understanding patterns, trends, and anomalies.
- Common visualizations include line plots, scatter plots, seasonal decomposition plots, autocorrelation plots, and heatmaps.
    - Visualizations provide insights into the data's behavior and guide further analysis.

Time Series Analysis Techniques:

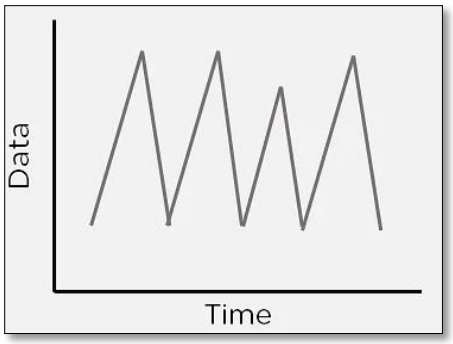
TSA employs a range of statistical techniques, including:

- Descriptive Analysis: Summarizing the data using measures such as mean, median, standard deviation, and percentiles.
- Autocorrelation Analysis: Examining the correlation between a time series and its lagged values to identify dependencies.
- Forecasting: Using past observations to predict future values of the time series.
- Time Series Modeling: Building mathematical models to capture the underlying patterns and relationships in the data.
- Seasonal Adjustment: Removing the seasonal component from the data to focus on the underlying trend and irregular components.
- TSA is applied in various domains, including finance, economics, marketing, weather forecasting, and resource allocation. It helps in understanding historical patterns, making future predictions, and guiding decision-making processes.

Working of Time Series Analysis

Sometimes data changes over time. This data is called time-dependent data. Given time-dependent data, you can analyze the past to predict the future. The future prediction will also include time as a variable, and the output will vary with time. Using time-dependent data, you can find patterns that repeat over time.

A Time Series is a set of observations that are collected after regular intervals of time. If plotted, the Time series would always have one of its axes as time.



Time Series Analysis collects data over time.

Data cleaning

Data cleaning is the process of identifying and correcting inaccurate records from a dataset along with recognizing unreliable or irrelevant parts of the data.

Handling Missing Values:

- Identify missing values in the time series data.
- Decide on an appropriate method to handle missing values, such as interpolation, forward filling, or backward filling.
- Use pandas or other libraries to fill or interpolate missing values.

Outlier Detection and Treatment:

- Identify outliers in the time series data that may be caused by measurement errors or anomalies.
- Use statistical techniques, such as z-score or modified z-score, to detect outliers.
- Decide on the treatment of outliers, such as removing them, imputing them with a reasonable value, or replacing them using smoothing techniques.

Handling Duplicates:

- Check for duplicate entries in the time series data.
- Remove or handle duplicate values appropriately based on the specific requirements of the analysis.

Resampling and Frequency Conversion:

- Adjust the frequency of the time series data if needed.
- Convert the data to a lower frequency (e.g., from daily to monthly) or a higher frequency (e.g., from monthly to daily) based on the analysis requirements.
- Use functions like resample() in pandas to perform resampling.

Addressing Non-Stationarity:

- Check for non-stationarity in the time series data, which can affect the analysis results.
- Apply techniques like differencing to make the data stationary, which involves computing the differences between consecutive observations.
- Use statistical tests like the Augmented Dickey-Fuller (ADF) test to test for stationarity.

Handling Time Zones and Daylight Saving Time:

- Ensure that the time series data is in the correct time zone and accounts for daylight saving time if applicable.
- Adjust the timestamps accordingly to maintain consistency in the data.

Consistent Time Intervals:

- Verify that the time series data has consistent and regular time intervals.
- Fill any gaps or irregularities in the time intervals, if necessary.

Simple python program to explain missing values in time series data

```
import pandas as pd
import numpy as np

# Create a sample time series dataset with missing values
data = {'Date': ['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',
'2021-01-05'],
        'Value': [10, np.nan, 12, 18, np.nan]}

# Convert the dictionary to a pandas DataFrame
df = pd.DataFrame(data)

# Convert the 'Date' column to datetime format
df['Date'] = pd.to_datetime(df['Date'])

# Set the 'Date' column as the index
df.set_index('Date', inplace=True)

# Handling missing values
df_filled = df.fillna(method='ffill')  # Forward fill missing values
df_interpolated = df.interpolate()  # Interpolate missing values

# Print the original and cleaned time series data
print("Original Data:\n", df)
```

```
print("\nForward Filled Data:\n", df_filled)
print("\nInterpolated Data:\n", df_interpolated)
```

Output

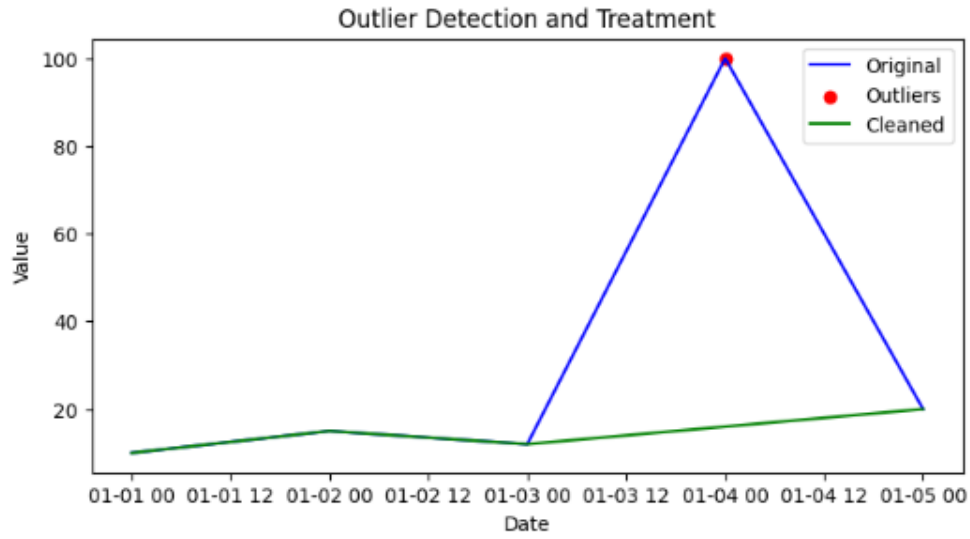
```
Original Data:
             Value
Date
2021-01-01   10.0
2021-01-02    NaN
2021-01-03   12.0
2021-01-04   18.0
2021-01-05    NaN

Forward Filled Data:
             Value
Date
2021-01-01   10.0
2021-01-02   10.0
2021-01-03   12.0
2021-01-04   18.0
2021-01-05   18.0

Interpolated Data:
             Value
Date
2021-01-01   10.0
2021-01-02   11.0
2021-01-03   12.0
2021-01-04   18.0
2021-01-05   18.0
```

Simple python program to detect outliers in time series data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Create a sample time series dataset with outliers
data = {'Date': ['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',
'2021-01-05'],
        'Value': [10, 15, 12, 100, 20]}

# Convert the dictionary to a pandas DataFrame
df = pd.DataFrame(data)
```

```
# Convert the 'Date' column to datetime format
df['Date'] = pd.to_datetime(df['Date'])

# Set the 'Date' column as the index
df.set_index('Date', inplace=True)

# Detect outliers using a threshold
threshold = 30  # Set the threshold for outlier detection
outliers = df[df['Value'] > threshold]

# Remove outliers
df_cleaned = df[df['Value'] <= threshold]

# Plot the original and cleaned time series data with outliers highlighted
plt.figure(figsize=(8, 4))
plt.plot(df.index, df['Value'], label='Original', color='blue')
plt.scatter(outliers.index, outliers['Value'], color='red', label='Outliers')
plt.plot(df_cleaned.index, df_cleaned['Value'], label='Cleaned',
color='green')
plt.xlabel('Date')
plt.ylabel('Value')
plt.title('Outlier Detection and Treatment')
plt.legend()
plt.show()

# Print the original and cleaned time series data
print("Original Data:\n", df)
print("\nCleaned Data:\n", df_cleaned)
```

Output




```
Original Data:
             Value
Date
```

```
2021-01-01     10
2021-01-02     15
2021-01-03     12
2021-01-04    100
2021-01-05     20

Cleaned Data:
             Value
Date
2021-01-01     10
2021-01-02     15
2021-01-03     12
2021-01-05     20
```

Time-based indexing

Time-based indexing refers to the process of organizing and accessing data based on timestamps or time intervals. It involves assigning timestamps to data records or events and utilizing these timestamps to efficiently retrieve and manipulate the data.

In time-based indexing, each data record or event is associated with a timestamp indicating when it occurred. The timestamps can be precise points in time or time intervals, depending on the granularity required for the application. The data is then organized and indexed based on these timestamps, enabling quick and efficient access to specific time ranges or individual timestamps.

Time-based indexing is commonly used in various domains that involve time-series data or events, such as financial markets, scientific research, IoT (Internet of Things) applications, system monitoring, and social media analysis.

In the context of TSA (Time Series Analysis), time-based indexing refers to the practice of organizing and accessing time-series data based on the timestamps associated with each observation. TSA involves analyzing and modeling data that is collected over time, and time-based indexing plays a crucial role in effectively working with such data.

Time-based indexing allows for efficient retrieval and manipulation of time-series data, enabling various operations such as subsetting, filtering, and aggregation based on specific time periods or intervals.

In TSA, time-based indexing is typically implemented using specialized data structures or libraries that provide functionality for working with time-series data. Some popular libraries for time-based indexing and analysis in Python include:

- *Pandas:* Pandas provides the DateTimeIndex object, which allows for indexing and manipulation of time-series data. It offers a wide range of time-based operations, such as slicing by specific time periods, resampling at different frequencies, and handling missing or irregular timestamps.

- *Statsmodels:* Statsmodels is a Python library that includes extensive functionality for time-series analysis. It provides time-series models and statistical tools for various types of time-based data analysis. It works well with Pandas' DateTimeIndex and provides methods for model estimation, forecasting, and diagnostics.

- *NumPy*: Although not specifically designed for time-series analysis, NumPy, a fundamental library for numerical computations in Python, can be used for time-based indexing. NumPy's array indexing and slicing capabilities, combined with timestamps represented as numerical values, allow for efficient retrieval and manipulation of time-series data.

Time-based indexing in TSA is essential for conducting exploratory data analysis, fitting time-series models, forecasting future values, and evaluating model performance.

Time-based indexing operations

Here are some common time-based indexing operations:

Slicing: Slicing involves retrieving a subset of data within a specific time range. With time-based indexing, you can easily slice the time-series data based on specific dates, times, or time intervals.

Example:

```
# Retrieve data between two specific dates
subset = df['2023-01-01':'2023-03-31']
```

Simple Python Program to Explain Time Series Slicing

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'],
    'value': [10, 15, 12, 18]
}

df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
```

```
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Retrieve data between two specific dates
subset = df['2023-01-02':'2023-01-03']
print(subset)
```

Output

```
            value
timestamp
2023-01-02     15
2023-01-03     12
```

Resampling: Resampling involves changing the frequency of the time-series data. You can upsample (increase frequency) or downsample (decrease frequency) the data to different time intervals, such as aggregating hourly data to daily data or converting daily data to monthly data.

Example:

```
# Resample data to monthly frequency
monthly_data = df.resample('M').mean()
```

Simple Python Program to Explain Resampling

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'],
    'value': [10, 15, 12, 18]
}

df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Resample data to monthly frequency
monthly_data = df.resample('M').mean()
print(monthly_data)
```

Output

```
             value
timestamp
2023-01-31  13.75
```

Shifting: Shifting involves moving the timestamps of the data forwards or backwards by a specified number of time units. This operation is useful for calculating time differences or creating lagged variables.

Example:

```
# Shift the data one day forward
shifted_data = df.shift(1, freq='D')
```

Simple Python Program to Explain Shifting

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'],
    'value': [10, 15, 12, 18]
}

df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Shift the data one day forward
shifted_data = df.shift(1, freq='D')
print(shifted_data)
```

Output

```
             value
timestamp
2023-01-02     10
2023-01-03     15
2023-01-04     12
2023-01-05     18
```

Rolling Windows: Rolling windows involve calculating statistics over a moving window of data. It allows for analyzing trends or patterns in a time-series by considering a fixed-size window of observations.
Example:

```
# Calculate the rolling average over a 7-day window
rolling_avg = df['value'].rolling(window=7).mean()
```

Simple Python Program to Explain Rolling Windows

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'],
    'value': [10, 15, 12, 18]
}

df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Calculate the rolling average over a 2-day window
rolling_avg = df['value'].rolling(window=2).mean()
print(rolling_avg)
```

Output

```
timestamp
2023-01-01     NaN
2023-01-02    12.5
2023-01-03    13.5
2023-01-04    15.0
Name: value, dtype: float64
```

Grouping and Aggregation: Grouping and aggregation operations involve grouping the time-series data based on specific time periods (e.g., days, weeks, months) and performing calculations on each group, such as calculating the sum, mean, or maximum value.

Example:

```
# Calculate the sum of values for each month
monthly_sum = df.groupby(pd.Grouper(freq='M')).sum()
```

Simple Python Program to Explain Grouping and Aggregation

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'],
    'value': [10, 15, 12, 18]
}

df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Calculate the sum of values for each month
monthly_sum = df.groupby(pd.Grouper(freq='M')).sum()
print(monthly_sum)
```

Output

```
            value
timestamp
2023-01-31     55
```

Time based indexing using pandas

A pandas DataFrame or Series with a time-based index is defined as a time series. The parameters in the time series can be anything that can fit inside the containers. Date or time values are merely used to retrieve them. In pandas, a time series container may be altered in a variety of ways.

Simple Example for time-based indexing

```
import pandas as pd

# Create a sample time-series dataset
data = {
    'timestamp': ['2023-01-01', '2023-02-01', '2023-03-01', '2023-04-01'],
    'value': [10, 15, 12, 18]
}
```

```
df = pd.DataFrame(data)

# Convert 'timestamp' column to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Set 'timestamp' as the index
df.set_index('timestamp', inplace=True)

# Access data for a specific time period
subset = df['2023-02-01':'2023-03-01']
print(subset)

# Resample the data to a different frequency
resampled_data = df.resample('1M').mean()
print(resampled_data)
```

Output

```
            value
timestamp
2023-02-01     15
2023-03-01     12
            value
timestamp
2023-01-31   10.0
2023-02-28   15.0
2023-03-31   12.0
2023-04-30   18.0
```

In the example above, we start by creating a DataFrame with a 'timestamp' column and a corresponding 'value' column. We convert the 'timestamp' column to a datetime data type using pd.to_datetime(). Next, we set the 'timestamp' column as the index using set_index().

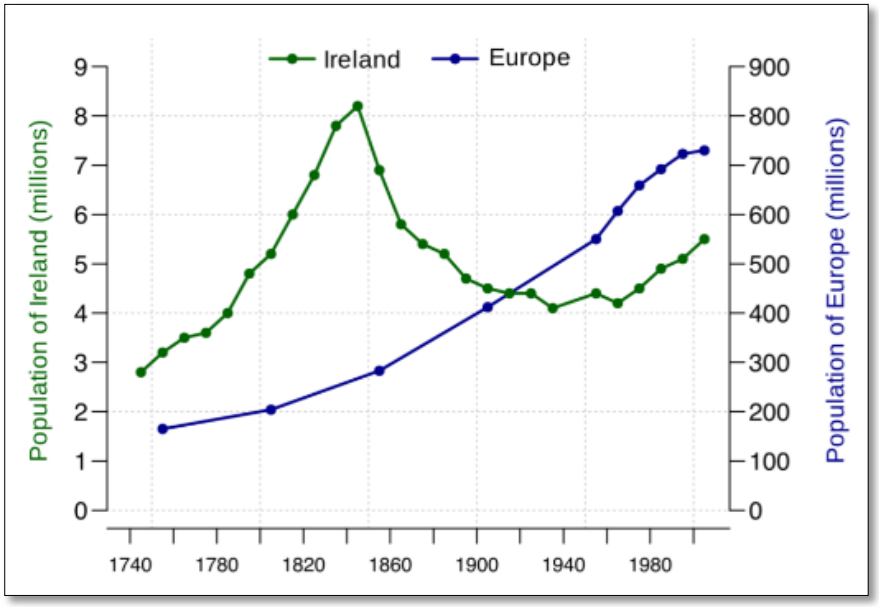
We then demonstrate two common time-based indexing operations. First, we access a subset of the data for a specific time period using time-based slicing with df['start_date':'end_date']. In this case, we retrieve the data between February 1, 2023, and March 1, 2023.

Next, we showcase resampling the data to a different frequency using resample(). We specify '1M' as the frequency, indicating monthly resampling. The data is resampled by taking the mean value for each month.

Visualizing Time Series Data

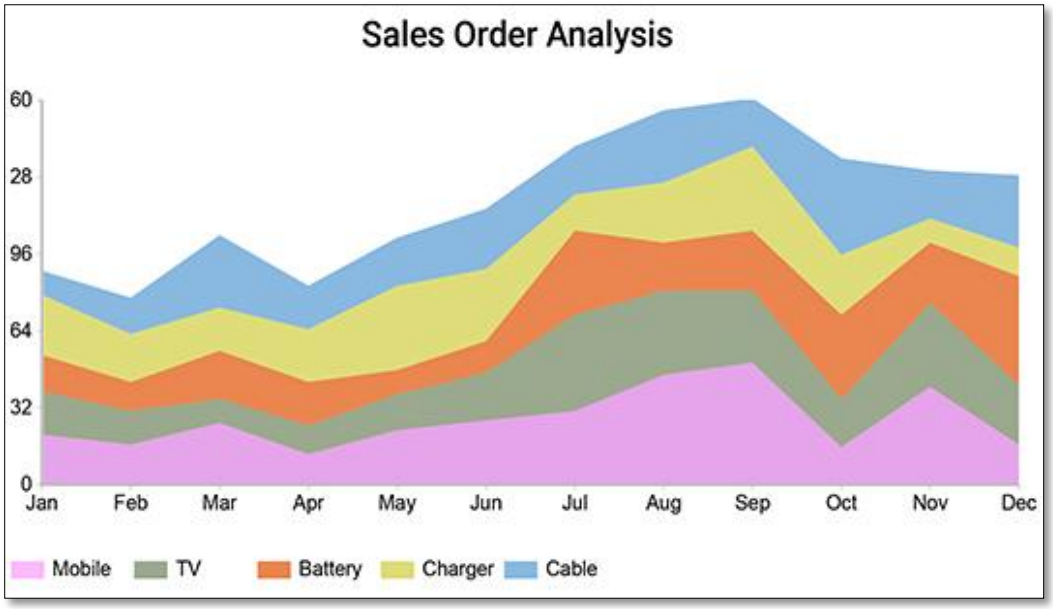
Line graph

A line graph is a common and effective way to visualize time series data. It displays data points connected by straight lines, allowing you to observe the trend and changes in values over time



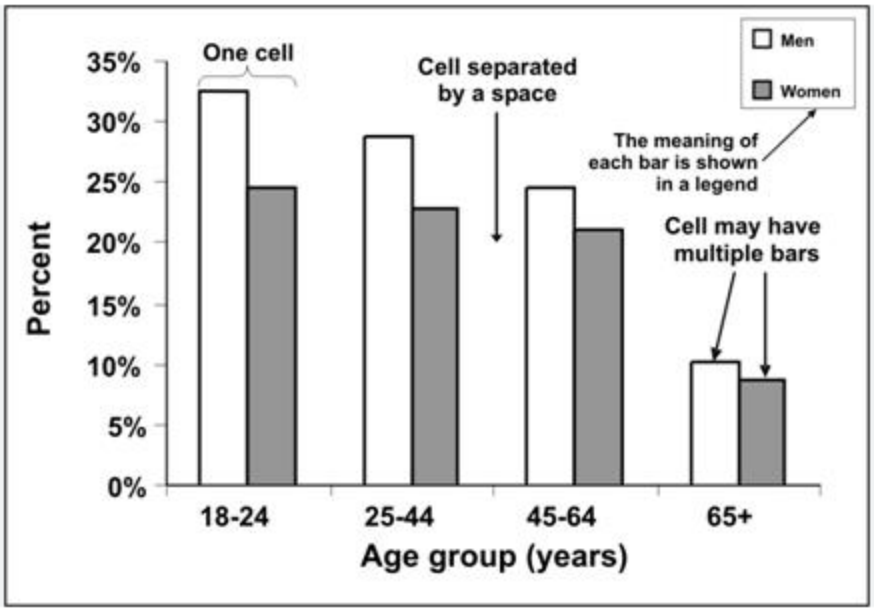
Stacked Area Chart

A stacked area chart, also known as a stacked area graph, is a type of graph used to visualize the cumulative contribution or proportion of multiple variables over time. It displays multiple series as stacked areas, where the height of each area represents the value of a particular variable at a given time. The areas are stacked on top of each other, illustrating how the variables contribute to the total value or the overall trend.



Bar Charts

Bar charts, also known as bar graphs or column charts, are a type of graph that uses rectangular bars to represent data. They are widely used for visualizing categorical or discrete data, where each category is represented by a separate bar. Bar charts are effective in displaying comparisons between different categories or showing the distribution of a single variable across different groups. The length of each bar is proportional to the value of the variable at that point in time.



Gantt chart

A Gantt chart is a type of bar chart that is commonly used in project management to visually represent project schedules and tasks over time. It provides a graphical representation of the project timeline, showing the start and end dates of tasks, as well as their duration and dependencies.

The key features of a Gantt chart are as follows:

- Task Bars: Each task is represented by a horizontal bar on the chart. The length of the bar indicates the duration of the task, and its position on the chart indicates the start and end dates.
- Timeline: The horizontal axis of the chart represents the project timeline, typically displayed in increments of days, weeks, or months. It allows for easy visualization of the project duration and scheduling.
- Dependencies: Gantt charts often include arrows or lines between tasks to represent dependencies or relationships between them. This helps to visualize the order in which tasks need to be completed and identify any critical paths or potential

bottlenecks.

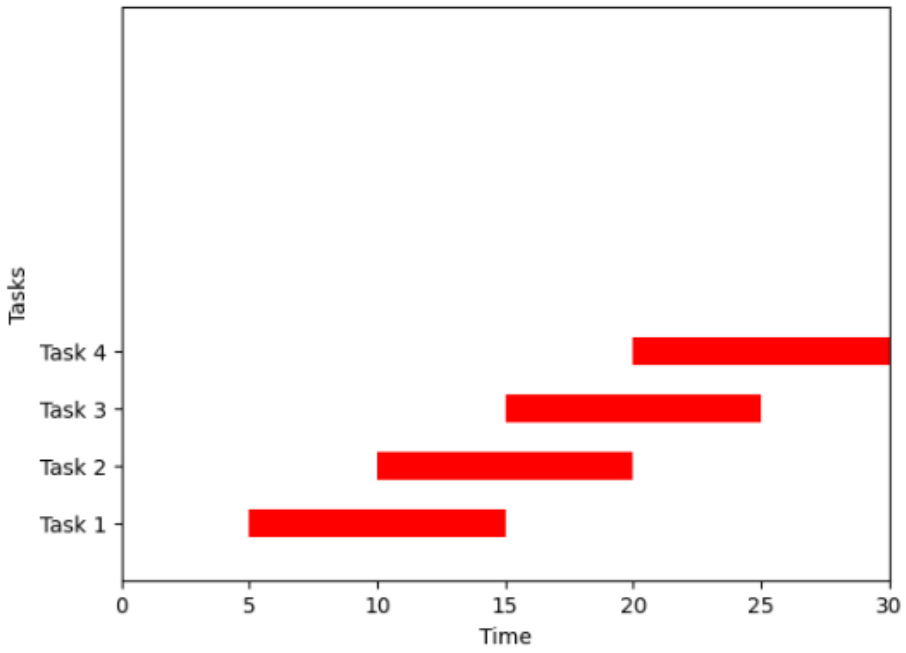
Milestones: Milestones are significant events or achievements within a project. They are typically represented by diamond-shaped markers on the chart to indicate important deadlines or deliverables.

Simple Python Program to Explain Gnatt Chart

```
def create_gantt_chart(tasks):
    fig, ax = plt.subplots()

    # Set y-axis limits
    ax.set_ylim(0, 10)

    # Set x-axis limits and labels
    ax.set_xlim(0, 30)
    ax.set_xlabel('Time')
    ax.set_ylabel('Tasks')

    # Plot the tasks as horizontal bars
    for task in tasks:
        start = task['start']
        end = task['end']
        y = task['task_id']
        ax.barh(y,   end   -   start,   left=start,   height=0.5,
align='center', color='red')

    # Set the y-ticks and labels
    y_ticks = [task['task_id'] for task in tasks]
    y_labels = [task['name'] for task in tasks]
    ax.set_yticks(y_ticks)
    ax.set_yticklabels(y_labels)

    # Display the Gantt chart
    plt.show()

# Example tasks data
tasks = [
    {'name': 'Task 1', 'start': 5, 'end': 15, 'task_id': 1},
    {'name': 'Task 2', 'start': 10, 'end': 20, 'task_id': 2},
    {'name': 'Task 3', 'start': 15, 'end': 25, 'task_id': 3},
    {'name': 'Task 4', 'start': 20, 'end': 30, 'task_id': 4}
]
```

```
# Create the Gantt chart
create_gantt_chart(tasks)
```

Output



Stream graph

A stream graph is a variation of a stacked area chart that displays changes in data over time of different categories through the use of flowing, organic shapes that create an aesthetic river/stream appearance. Unlike the stacked area chart, which plots data over a fixed, straight axis, the stream plot has values displaced around a varying central baseline.

Each individual stream shape in the stream graph is proportional to the values of it's categories. Color can be used to either distinguish each category or to visualize each category's additional quantitative values through varying the color shade.

Making a Stream Graph with Python

For this example we will use Altair, which is a graphing library in python. Altair is a declarative statistical visualization library, based on Vega and Vega-Lite. The source code is available on GitHub.

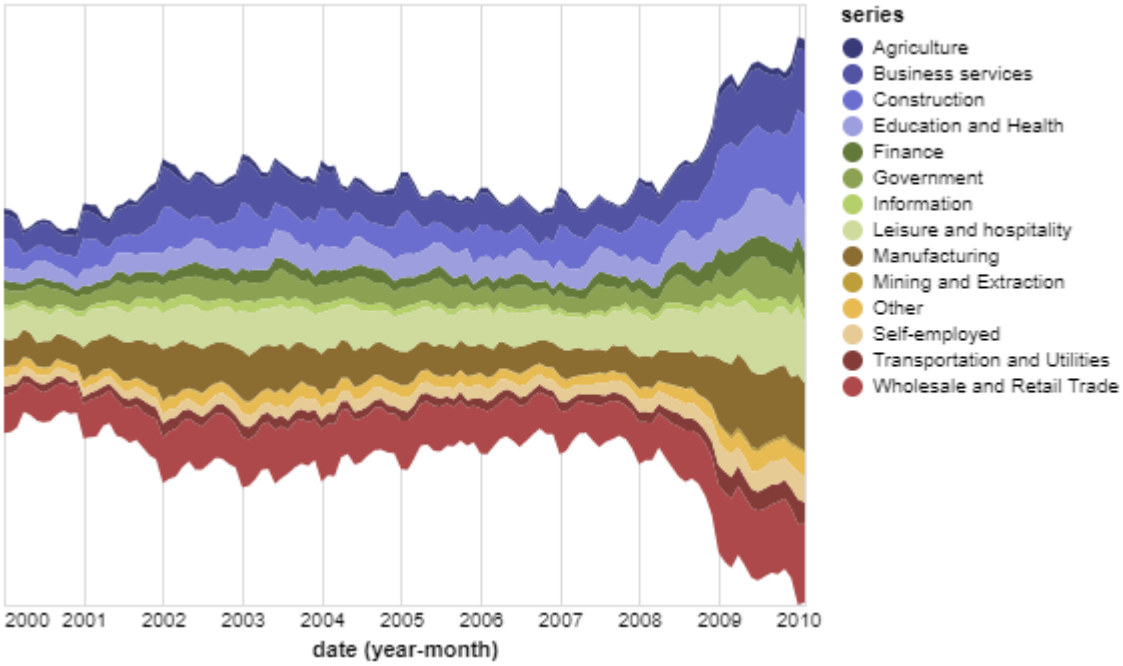
To begin creating our stream graph, we will need to first install Altair and vega_datasets.

```
!pip install altair
!pip install vega_datasets
```

Now, let's use altair and the vega datasets to create an interactive stream graph looking at unemployment data across a series of 10 years of time across multiple industries.

```
import altair as alt
from vega_datasets import data

source = data.unemployment_across_industries.url

alt.Chart(source).mark_area().encode(
    alt.X('yearmonth(date):T',
        axis=alt.Axis(format='%Y', domain=False, tickSize=0)
    ),
    alt.Y('sum(count):Q', stack='center', axis=None),
    alt.Color('series:N',
        scale=alt.Scale(scheme='category20b')
    )
).interactive()
```

Output



Heat map

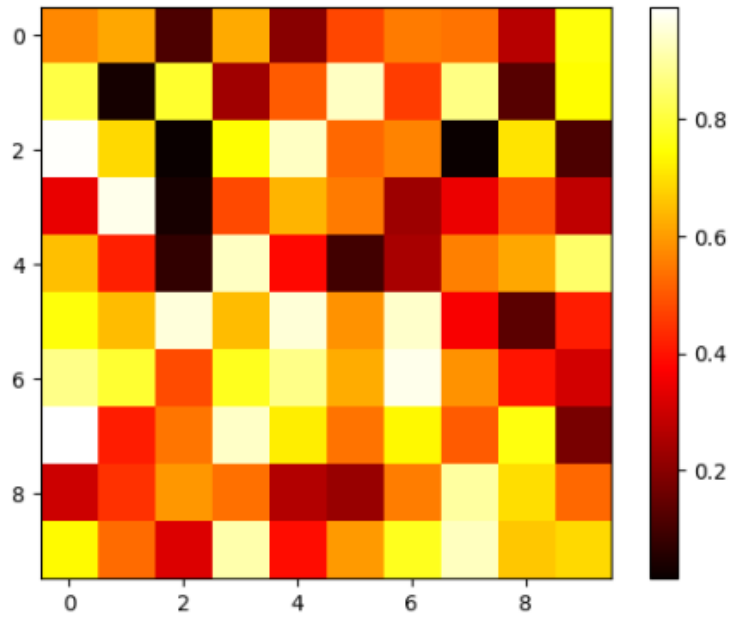
A heat map is a graphical representation of data where individual values are represented as colors. It is typically used to visualize the density or intensity of a particular phenomenon over a geographic area or a grid of cells.

In a heat map, each data point is assigned a color based on its value or frequency. Typically, a gradient of colors is used, ranging from cooler colors (such as blue or green) to warmer colors (such as yellow or red). The colors indicate the magnitude of the data, with darker or more intense colors representing higher values and lighter or less intense colors representing lower values.

Heat maps are commonly used in various fields, including data analysis, statistics, finance, marketing, and geographic information systems (GIS). They can provide insights into patterns, trends, or anomalies in the data by visually highlighting areas of higher or lower concentration.

```
import numpy as np
import matplotlib.pyplot as plt
# Generate random data
data = np.random.rand(10, 10)
# Create heatmap
plt.imshow(data, cmap='hot', interpolation='nearest')
# Add color bar
plt.colorbar()
# Show the plot
plt.show()
```

output



Grouping

Grouping time series data involves dividing it into distinct groups based on certain criteria. This grouping can be useful for performing calculations, aggregations, or analyses on specific subsets of the data. In Python, you can use the pandas library to perform grouping operations on time series data. Here's an example of how to group time series data using pandas:

Simple Python program to explain Grouping

```
import pandas as pd

# Create a sample DataFrame with a datetime index
data = {'date': pd.date_range(start='2022-01-01', periods=100,
freq='D'),
        'category': ['A', 'B', 'A', 'B'] * 25,
        'value': range(100)}
df = pd.DataFrame(data)
df.set_index('date', inplace=True)

# Grouping by category and calculating the sum
grouped_df = df.groupby('category').sum()

# Grouping by month and calculating the mean
monthly_df = df.groupby(pd.Grouper(freq='M')).mean()

# Print the grouped DataFrames
print("Grouped DataFrame (by category):")
print(grouped_df)

print("\nMonthly DataFrame (mean per month):")
print(monthly_df)
```

Output

```
Grouped DataFrame (by category):
          value
category
A          2450
B          2500

Monthly DataFrame (mean per month):
            value
date
2022-01-31   15.0
2022-02-28   44.5
2022-03-31   74.0
2022-04-30   94.5
```

In this example, we first create a sample DataFrame df with a datetime index. The DataFrame contains a 'value' column ranging from 0 to 99 and a 'category' column with two distinct categories 'A' and 'B'.

We then proceed with the grouping operations:

- To group by a specific column, such as 'category', we use the groupby() function, specifying the column name as the argument. In this case, we group the data by the 'category' column and calculate the sum for each group using .sum().
- To group by a specific time period, such as month, we can use the groupby() function with pd.Grouper(freq='M'). This creates a grouper object that can be used to group the data by the desired frequency. In this case, we group the data by month and calculate the mean value for each month using .mean().

Resampling

Resampling time series data involves grouping the data into different time intervals and aggregating or summarizing the values within each interval. This process is useful when you want to change the frequency or granularity of the data or when you need to perform calculations over specific time intervals. There are two common methods for resampling time series data: upsampling and downsampling.

Downsampling: Downsampling involves reducing the frequency of the data by grouping it into larger time intervals. This is typically done by aggregating or summarizing the data within each interval. Some common downsampling methods include:

- Mean/Median: Calculate the mean or median value within each interval.
- Sum: Calculate the sum of values within each interval.
- Min/Max: Determine the minimum or maximum value within each interval.
- Resample Method: Use specialized resampling methods like interpolation or forward/backward filling to estimate values within each interval.

Here's an example of downsampling time series data using the resample() function in pandas:

```
import pandas as pd

# Assuming 'df' is your DataFrame with a datetime index
downsampled_df = df.resample('D').mean()
```

In this example, the data is being downsampled to daily frequency, and the mean value within each day is calculated.

Upsampling: Upsampling involves increasing the frequency of the data by grouping it into smaller time intervals. This may require filling in missing values or interpolating to estimate values within the new intervals. Some common upsampling methods include:

- Forward/Backward Filling: Propagate the last known value forward or backward to fill missing values within each interval.
- Interpolation: Use interpolation methods like linear, polynomial, or spline interpolation to estimate values within each interval.
- Resample Method: Utilize specialized resampling methods to estimate values within each interval.

Here's an example of upsampling time series data using the resample() function in pandas:

```
import pandas as pd

# Assuming 'df' is your DataFrame with a datetime index
upsampled_df = df.resample('H').interpolate()
```

In this example, the data is being upsampled to hourly frequency, and missing values are interpolated using the interpolate() function.

Simple python program to explain upsampling and Downsampling

```
import pandas as pd

# Create a sample DataFrame with a datetime index
data = {'date': pd.date_range(start='2022-01-01', periods=100, freq='D'),
        'value': range(100)}
df = pd.DataFrame(data)
df.set_index('date', inplace=True)

# Downsampling to monthly frequency
downsampled_df = df.resample('M').mean()

# Upsampling to hourly frequency
upsampled_df = df.resample('H').interpolate()

# Print the downsampled and upsampled DataFrames
print("Downsampled DataFrame:")
print(downsampled_df.head())

print("\nUpsampled DataFrame:")
print(upsampled_df.head())
```

Output

```
Downsampled DataFrame:
             value
```

```
date
2022-01-31   15.0
2022-02-28   44.5
2022-03-31   74.0
2022-04-30   94.5

Upsampled DataFrame:
                         value
date
2022-01-01 00:00:00  0.000000
2022-01-01 01:00:00  0.041667
2022-01-01 02:00:00  0.083333
2022-01-01 03:00:00  0.125000
2022-01-01 04:00:00  0.166667
```

In this example, we first create a sample DataFrame df with a datetime index. The DataFrame contains a 'value' column ranging from 0 to 99, with a daily frequency for the 'date' index.

We then proceed with the resampling:

- For downsampling, we use the resample() function with the argument 'M' to downsample the data to monthly frequency. In this case, we calculate the mean value for each month using .mean().
- For upsampling, we use the resample() function with the argument 'H' to upsample the data to hourly frequency. We use .interpolate() to fill in the missing values within each hour using interpolation.