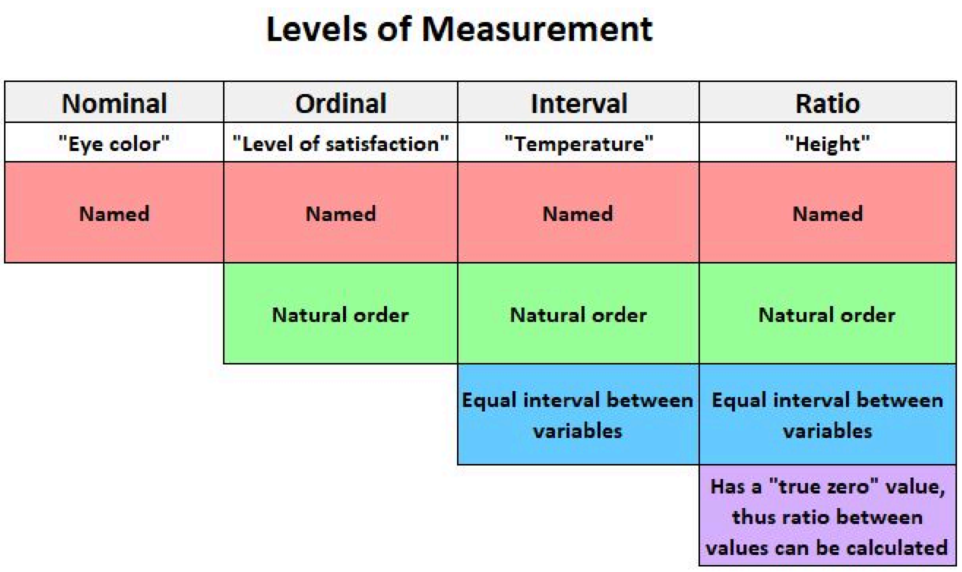
Module 1: Introduction, Probability, Monte Carlo Simulations, & Profiling

1-1 Data Types:

Data can be classified based on the levels of measurement: **Nominal, Ordinal, Interval, and Ratio.**

- **Nominal Data:** unique identifiers but no distinct order (i.e. names, colors)
- **Ordinal Data:** unique identifiers and distinct order (i.e. letter grades, military ranks)
- **Continuous Data, Interval:** numerical values with meaningful differences, but no distinct 'true zero' to construct ratios from (i.e. temperature in $^\circ F$/ $^\circ C$, days in calendar, credit scores)
- **Continuous Data, Ratio:** numerical values with meaningful differences, and a distinct 'true zero' to construct ratios from (i.e. temperature in $^\circ K$, salaries, weights, distances)



Levels of Measurement

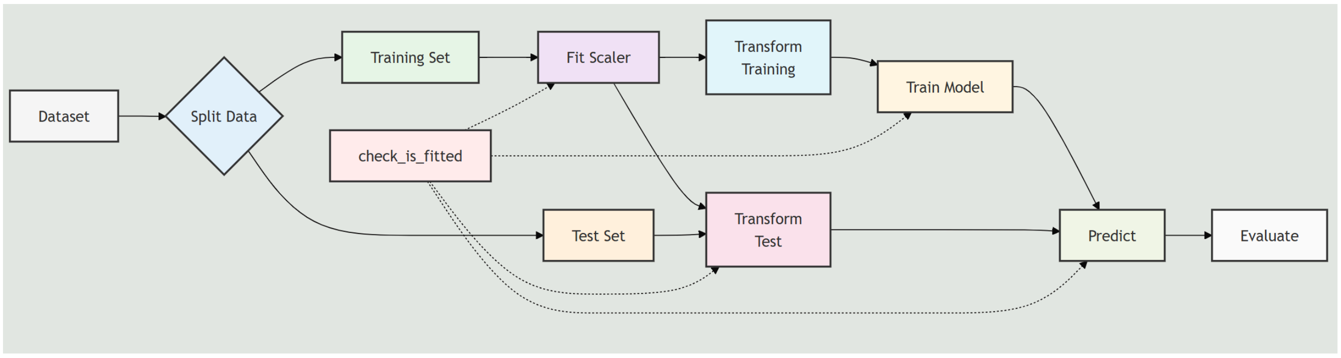
Nominal	Ordinal	Interval	Ratio
"Eye color"	"Level of satisfaction"	"Temperature"	"Height"
Named	Named	Named	Named
	Natural order	Natural order	Natural order
		Equal interval between variables	Equal interval between variables
			Has a "true zero" value, thus ratio between values can be calculated

1-2 Training vs Testing Data:

When using models to make predictions, **Data Leakage** can become a serious problem, with the model learning on data of which it is also tested on, which will artificially inflate accuracy metrics as the model memorizes the data it is trained on rather than learn the underlying relationships present in the data, also known as **Over-fitting**.

To fix this, we make a clean split in the dataset, creating separate **Training Data** and **Testing Data**. Thus, during preprocessing (such as scaling) we have to make sure that the no information about the **Training Data** leaks into the **Testing Data**.

This manifests by both **Fitting** and **Transforming** on the **Training Data** while only **Transforming** on the **Testing Data**. This prevents information about the testing data such as its mean from being incorporated into the training data while ensuring both sets of data are scaled identically.



Fitting is distinguished by the storage of data used in the **Transform** operation, such as the mean and standard deviation of the data.

1-3 Scaling:

When dealing with datasets that have features of varying magnitudes, **Scaling** is incredibly important. **Scaling** your data will put all of the features on a common scale and will help prevent features with a large order of magnitude from dominating your model. We will most commonly encounter **Standard Scaling** and **Min-Max Scaling**:

- **Standard Scaling**: Transforms the data to have a mean of 0 and a standard deviation of 1
- **Min-Max Scaling**: Transforms the data to have a minimum of 0 and a maximum of 1

Standard Scaling is the most commonly used, but **Min-Max Scaling** is very useful when the data is intended to be interpreted on a clear interval, such as grades from $0 - 100$.

Example: Construct a class of which implements **Standard Scaling**.

```
import numpy as np
from sklearn.model_selection import train_test_split

class StandardScaler:
    def __init__(self):
        self.mean_ = None
        self.scale_ = None
        self.is_fitted_ = False

    def fit(self, X):
        #Learn scaling parameters from training data, fit method stores information
        X = np.array(X)
        self.mean_ = np.mean(X, axis=0)
        self.scale_ = np.std(X, axis=0)
        self.is_fitted_ = True #Let object know it has been fit
        return self

    def transform(self, X):
        #Apply transformation
        X = np.array(X)
        return (X - self.mean_)/self.scale_

    def fit_transform(self, X):
        #Fit and transform in one step
        return self.fit(X).transform(X)

#Load In Data
X = 10*np.random.rand(100, 10) + 3 #Put your feature data (samples, features)
y = 10*np.random.rand(100) #Put your target data (samples,)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

#Create Scaler Object
scaler = StandardScaler()

#Scale Training and Testing Data
Xscaled_train = scaler.fit_transform(X_train)
Xscaled_test = scaler.transform(X_test)

#Print Results
print(f'Training Mean: {Xscaled_train.mean():.4f}')
print(f'Training Std:  {Xscaled_train.std():.4f}')
print(f'Training Mean: {Xscaled_test.mean():.4f}')
print(f'Testing Std:   {Xscaled_test.std():.4f}')
```

Output:

```
Training Mean: -0.0000
Training Std:  1.0000
Training Mean: -0.1270
Testing Std:   1.0394
```

1-4 Classes:

In Python, we often want to organize our code into **Classes** to assist with readability and organization. In general, most code should be organized in either **Classes** or **Functions**. **Classes** define **Attributes** and **Methods** to allow for easier access of variables and functions. In addition, there exist special **Dunder Methods** which define the behavior of the **Class** when Python operators are applied to it, such as `__add__` , `__str__` , `__len__` , `__eq__` , and `__init__` . (To see all such methods, call `dir(MyClass)`)

Example: Construct a basic example **Class**.

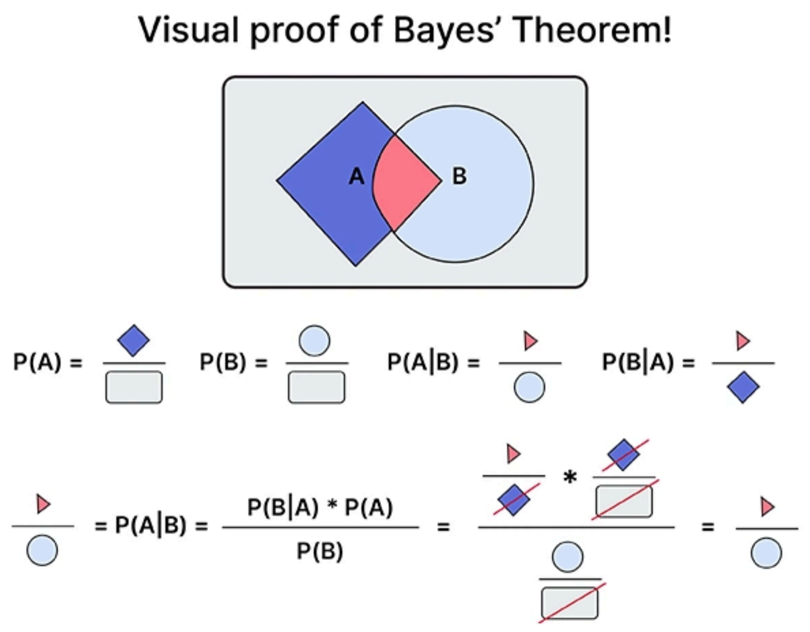
```
class MyClass:

    #Initialization statement, is run when creating a new instance of the class
    def __init__(self, arg1, arg2, kwarg1=True):
        #Defining attributes for each initialization argument
        self.arg1 = arg1
        self.arg2 = arg2
        self.kwarg1 = kwarg1

    def __add__(self, other):
        #Other denotes the other class instance being added
        return self.arg1 + other.arg1

    def mymethod(self):
        return (self.arg1)**2 + (self.arg2)**2


#Create two class instances
Object1 = MyClass(arg1=1, arg2=2) #__init__ is called
Object2 = MyClass(arg1=2, arg2=4) #__init__ is called

#Add them
print(f'Sum of Objects: {Object1 + Object2}') #__add__ is called

#Execute the defined method
print(f'mymethod result: {Object2.mymethod()}') #mymethod is called
```

Output:

```
Sum of Objects: 3
mymethod result: 20
```

1-5 Bayes Theorem:

Bayes Theorem allows us to describe the probability of an event given prior knowledge of the conditions related to the event. An example would be quantifying the probability that an individual has a disease provided the symptoms the have and information about the relation between the symptoms and the disease.


Visual proof of Bayes' Theorem!
A
B
P(A) =
P(B) =
P(A|B) =
P(B|A) =
= P(A|B) = P(B|A) * P(A) / P(B) =
*
=

The following two equations are differing forms of **Bayes Theorem**:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^C)P(A^C)}$$

Example: Say that there is a disease (event A) which affects 10% of the population and a symptom (event B) which affects 5% of the total population and 40% of the diseased population. What is the probability that an individual with this symptom has the disease?

$$P(A|B) = \frac{40\% * 10\%}{5\%} = 80\%$$

Code Implementation:

```
import numpy as np

def BayesDisease(num_samples=100_000):
    '''Determines the probability of having a disease given that you have a common symptom'''
    #Define Probabilities
    P_disease = 0.1 #10% of population has disease
    P_symptom = 0.05 #5% of population has symptom
    P_symptom_given_disease = 0.4 #40% of diseased people

    #Calculate P(symptom|no disease) or P(B|A^C) Using Intersection Probability
    #P(B|A^C) = (P(B) - P(A and B))/P(A^C) = (P(B) - P(B|A)*P(A))/P(A^C)
    P_symptom_given_no_disease = (P_symptom - P_symptom_given_disease * P_disease)/(1 - P_disease)

    #First, Construct One of The Properties (Disease or Symptom) Independently
    disease = np.random.rand(num_samples) < P_disease

    #Next, Construct The Other Property Naively (Assuming the first property is all true or all false)
    symptom = np.random.rand(num_samples) < P_symptom_given_disease #Assumes that everyone has the disease
    symptom[~disease] = np.random.rand(np.sum(~disease)) < P_symptom_given_no_disease #Corrects for those without the disease

    #Calculate P(Disease|Symptom)
    P_disease_given_symptom = np.sum(disease & symptom)/np.sum(symptom)

    return P_disease_given_symptom


num_samples = 100_000
P_disease_given_symptom = BayesDisease(num_samples=num_samples)
print(f'P(Disease|Symptom) with {num_samples} samples: {100*P_disease_given_symptom:.2f}%')
```

Output:

```
P(Disease|Symptom) with 100000 samples: 80.80%
```

Another useful equation would be for relating the intersection probability to the conditional probability:

$$P(A \cap B) = P(A)P(B|A) = P(B)P(A|B)$$

1-6 Monte Carlo Simulations:

Monte Carlo Simulations are a method to estimate probabilities by generating large numbers of samples through the **Law of Large Numbers**, which states that as the number of samples increase, the average probability will converge onto the theoretical probability.

There are two types of such convergence:

- **Weak Convergence:** The probability of large deviations from expectations goes to 0 as N approaches ∞. (Converges in Probability)
- **Strong Convergence:** Every outcome converging onto expectations as N approaches ∞. (Converges Almost Surely)

Example: Code a function to calculate the expected average sum of rolling 10 6-sided dice.

```
import numpy as np

def RollDice(N):
    '''Rolls 10 6-sided dice N times,
       Computes the sum of the values.
       The expected value for each sum is 35.'''
    vals = np.random.randint(1, 7, size=(10,N)) #Roll the dice
    sums = vals.sum(axis=0) #Compute the sums
    return sums
```

```
N = 10_000
sums = RollDice(N=N)
print(f'Average of Sums: {np.mean(sums)}') #Find the average value of the sums
```

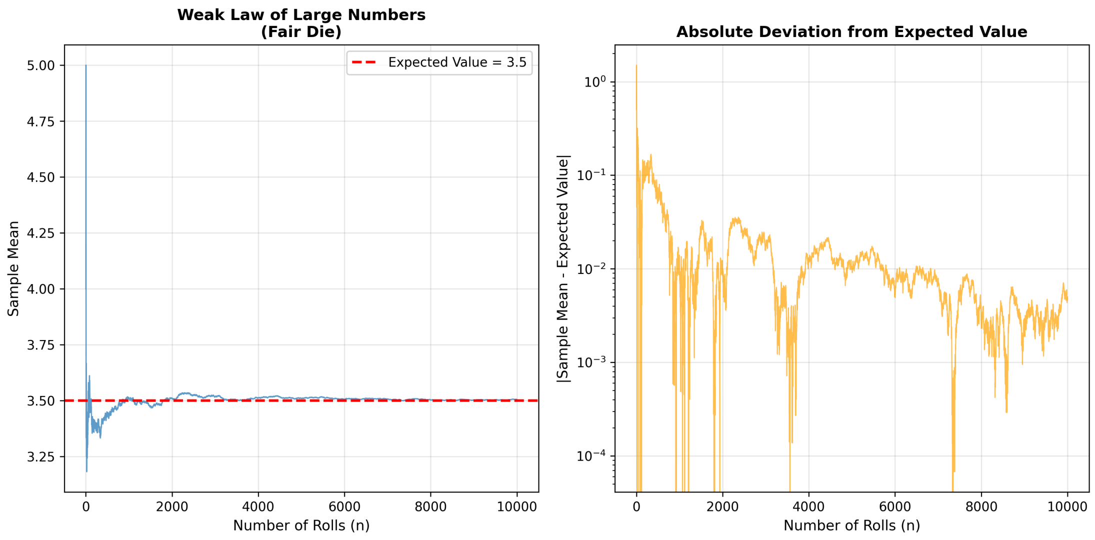
Output:

```
Average of Sums: 34.9733
```

The average sum of the sets of 10 rolls will exhibit **Strong Convergence** onto the expected sum of 35 as N approaches ∞. (i.e. guaranteed to converge)

An example of **Weak Convergence**:

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

# Parameters
max_samples = 10000
population_mean = 3.5  # Expected value of a fair die

# Simulate rolling a fair die
die_rolls = np.random.randint(1, 7, size=max_samples)

# Calculate cumulative mean at each step
sample_sizes = np.arange(1, max_samples + 1)
cumulative_means = np.cumsum(die_rolls) / sample_sizes

# Create visualization
plt.figure(figsize=(12, 6))

# Plot 1: Convergence of sample mean
plt.subplot(1, 2, 1)
plt.plot(sample_sizes, cumulative_means, linewidth=0.9, alpha=0.7)
plt.axhline(y=population_mean, color='r', linestyle='--',
            linewidth=2, label=f'Expected Value = {population_mean}')
plt.xlabel('Number of Rolls (n)', fontsize=11)
plt.ylabel('Sample Mean', fontsize=11)
plt.title('Weak Law of Large Numbers\n(Fair Die)', fontsize=12, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 2: Deviation from expected value
plt.subplot(1, 2, 2)
deviations = np.abs(cumulative_means - population_mean)
plt.plot(sample_sizes, deviations, linewidth=0.8, alpha=0.7, color='orange')
plt.xlabel('Number of Rolls (n)', fontsize=11)
plt.ylabel('|Sample Mean - Expected Value|', fontsize=11)
plt.title('Absolute Deviation from Expected Value', fontsize=12, fontweight='bold')
plt.grid(True, alpha=0.3)
plt.yscale('log')  # Log scale to better see convergence

plt.tight_layout()
plt.savefig('WeakLLN.png',dpi = 300)
plt.show()

# Print statistics at different sample sizes
print("Convergence Statistics:")
print("-" * 50)
for n in [10, 100, 1000, 10000]:
    sample_mean = cumulative_means[n-1]
    deviation = abs(sample_mean - population_mean)
    print(f"n = {n:5d}: Sample Mean = {sample_mean:.4f}, "
          f"Deviation = {deviation:.4f}")
```



```
Convergence Statistics:
--------------------------------------------------
n =    10: Sample Mean = 3.4000, Deviation = 0.1000
n =   100: Sample Mean = 3.4600, Deviation = 0.0400
n =  1000: Sample Mean = 3.5130, Deviation = 0.0130
n = 10000: Sample Mean = 3.5050, Deviation = 0.0050
```

1-7 PDFs & CDFs:

A statistical distribution can be modelled by both its **Probability Density Function (PDF)** and its **Cumulative Distribution Function (CDF)**.

PDFs describe the relative magnitude of the probability that the random variable takes a specific value. The magnitude of a **PDF** at a given value is NOT the probability that the random variable takes that exact value, but rather a measure or RELATIVE probability. If the **PDF** of a random variable is double the magnitude at a value a when compared to a value b, then the probability that the random variable takes the value a is twice the probability that takes the value b.

PDFs Must abide by the following properties:

- **Non-Negativity:** The function MUST be non-negative at all possible values.
- **Normalization:** The area under the function, which is the total probability of all outcomes, must equal 1.

CDFs describe the cumulative probability up to a given point, or the probability that the random variable takes a value which is less than or equal to that point. As such, its magnitude can be directly interpreted as a probability, as opposed to a relative density. The probability that the random variable lies between a value a and a value b is as follows:

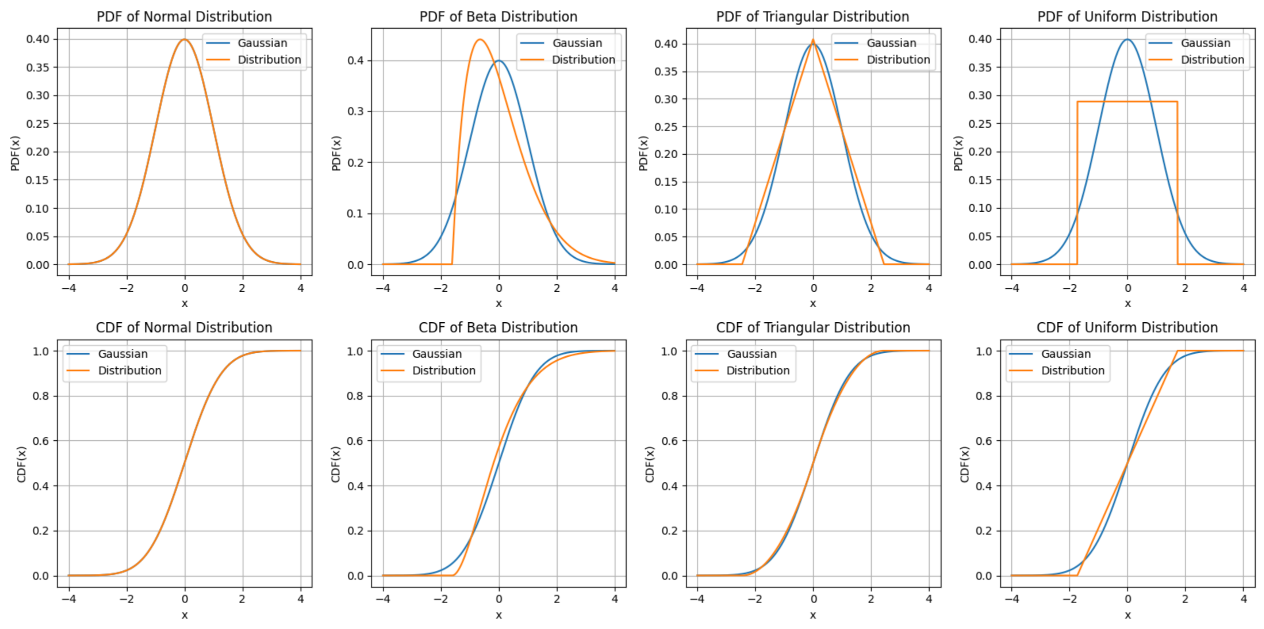
$$P(a < X \leq b) = F(b) - F(a)$$

Where F is the **CDF**, and a is less than b.

CDFs Must abide by the following properties:

- **Monotonicity:** The function MUST ALWAYS be non-decreasing from left to right.
- **Limits (CDF Normalization):** The function MUST approach 0 as $x \to -\infty$ and 1 as $x \to +\infty$.
- **PDF Relation:** The derivative of the **CDF** yields the **PDF**.

Below are some examples of common **PDFs** and **CDFs** (Generation code in the matching notebook):



1-8 Normality, Standardization, & QQ-Plots:

It is often useful to compare distributions in data, outcomes, and errors to a **Normal (Gaussian) Distribution**. In order to do this, we must **Standardize** our distribution (X) as follows:

$$Z = \frac{X - \mu}{\sigma}$$

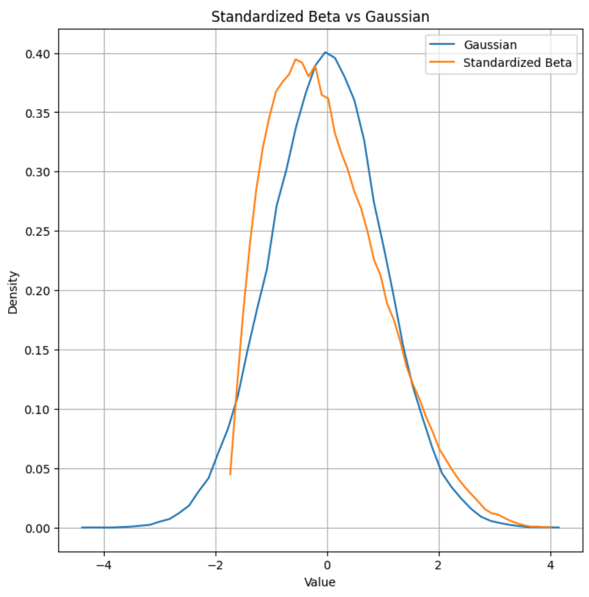
This sets the mean of out distribution to 0 and the standard deviation to 1, aligning it with the characteristics of a **Standard Gaussian Distribution**

As a quick visual test to check the **Normality**, or similarity to the **Standard Gaussian Distribution**, of the distribution we can construct a **QQ-Plot**.

Example: Plot the **PDF** of a Beta Distribution alongside the **PDF** of a **Standard Gaussian Distribution**.

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

#Generate The Samples
N = 100_000
Gaussian_Dist = stats.norm.rvs(loc=0, scale=1, size=N)
X = stats.beta.rvs(a=2, b=5, loc=10, scale=15, size=N)

#Standardize the Beta Distribution
Z = (X - np.mean(X))/np.std(X)

#Compute the Histogram Counts and Bin Edges
counts_Gaussian, bins_Gaussian = np.histogram(Gaussian_Dist, bins=50, density=True)
counts_Z, bins_Z = np.histogram(Z, bins=50, density=True)

#Compute the Centers of the Bins
bin_centers_Gaussian = 0.5*(bins_Gaussian[1:] + bins_Gaussian[:-1])
bin_centers_Z = 0.5*(bins_Z[1:] + bins_Z[:-1])

#Plot the Results
plt.figure(figsize=(8,8))
plt.plot(bin_centers_Gaussian, counts_Gaussian, label='Gaussian')
plt.plot(bin_centers_Z, counts_Z, label='Standardized Beta')

#Plot Formatting
plt.title('Standardized Beta vs Gaussian')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid()
plt.legend()
```

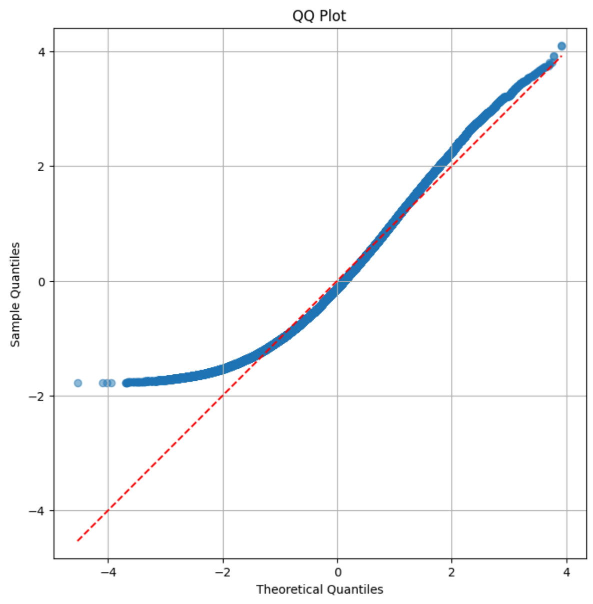
Output:



Example: Construct a standard **QQ-Plot** of the Beta Distribution.

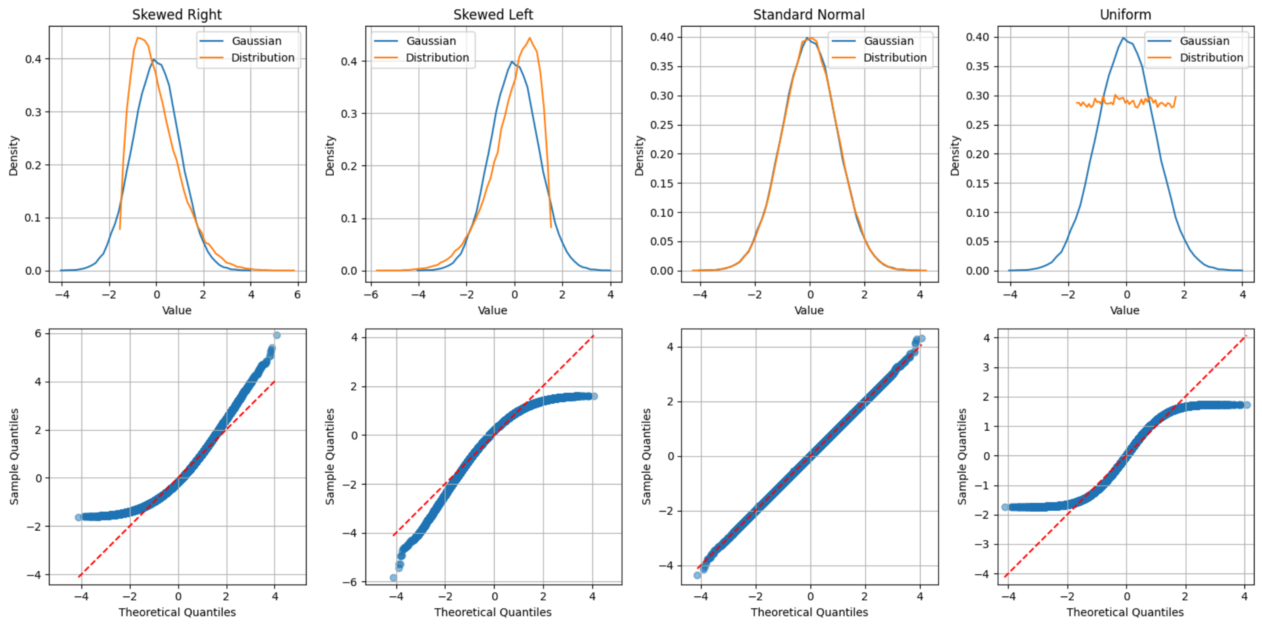
```
#Function to Generate QQ Plot
def QQplot(data):
    '''Generates a QQ plot comparing a dataset to a Standard Gaussian'''
    #Generate the Gaussian Data
    N = len(data)
    Gdata = stats.norm.rvs(loc=0, scale=1, size=N)

    #Sort the data
    sorted_Gdata = np.sort(Gdata)
    sorted_data = np.sort(data)

    #Generate the QQ plot and plot the data
    plt.figure(figsize=(8,8))
    plt.scatter(sorted_Gdata, sorted_data, alpha=0.5)

    #Plot y=x line for reference (Perfect Agreement)
    plt.plot([min(sorted_Gdata), max(sorted_Gdata)], [min(sorted_Gdata), max(sorted_Gdata)], color='red', linestyle='--')

    #Plot Formatting
    plt.title('QQ Plot')
    plt.xlabel('Theoretical Quantiles')
    plt.ylabel('Sample Quantiles')
    plt.grid()
    plt.axis('equal')
    plt.show()

#Construct the QQ-Plot for the Standardized Beta and Gaussian
QQplot(Z)
```

Output:



This **QQ-Plot** Shows that the distribution is **Skewed Right** and diverges heavily from the **Standard Gaussian** at the right tail. Note that **QQ-Plots** can be used both to measure similarity between a specified distribution and a standard distribution, as well as between two specified distributions, in which case the quantiles would not be labeled 'Theoretical' and 'Sample'.

Below are some examples of common **QQ-Plots** and how to interpret them (Generation code in the matching notebook):



1-9 Normality Tests:

To quantify the **Normality** metric shown above, we utilize **Normality Tests**, which compute a numerical difference metric between a specified distribution and the **Standard Gaussian** distribution. These tests can also be used, similarly to **QQ-Plots** to measure the difference between two specified distributions. We will review three main statistical **Normality Tests**:

1. **Kolmogorov Smirnov (KS) Test:** Computes the maximum vertical distance between the **CDFs** of the **Standard Gaussian** and our specified distribution.
2. **Anderson-Darling (AD) Test:** A modification of the **KS Test** which gives more weight to the tails of the distribution.
3. **Shapiro-Wilk (SW) Test:** Checks the correlation between the ordered samples, widely used for small to moderate sample sizes.

Each of these **Normality Tests** will output a **Test Statistic** of which can be used to calculate a **p-value**. If this value is less than a certain cutoff (typically 0.05), it suggests the distribution in question is not normal.

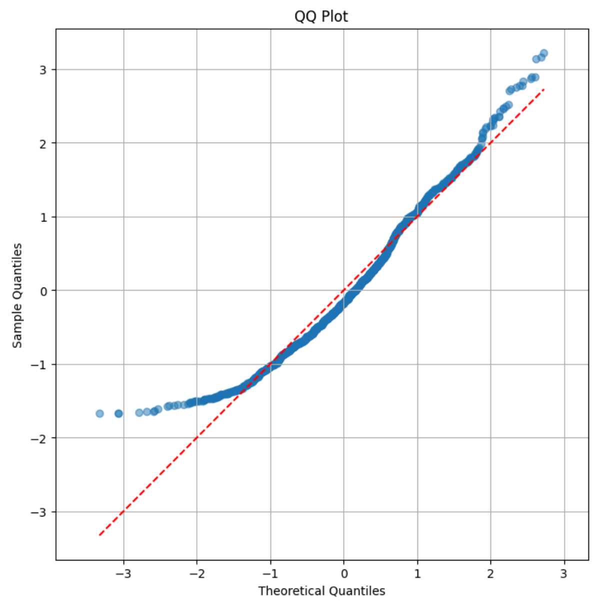
To use these tests, typically one must consider certain underlying assumptions:

- **Independence:** Observations in the sample must be independent (not causally related) of one another.
- **Continuity:** These tests are designed for **Continuous** data and cannot be used on **Categorical** or **Discrete** data.
- **Sample Size:** These tests are best used for sample sizes $20 - 2000$, with **SW** best used for $20 - 200$.
- **Sensitivity:** For high numbers of samples even tiny deviations from normality can cause rejection, always plot your data visually!

Example: Create a **Class** which implements the **KS**, **AD**, and **SW Normality Tests**. Plot the standardized samples on a **QQ-Plot**. Compute the **Test Statistics** and **p-values** for a specific distribution.

```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
from IPython.display import display

class NormalityTester:
    """Computes Test Statistics and p-values for KS, AD, and SW Normality Tests:"""
    def __init__(self, samples):
        #Standardize the samples
        self.samples = (samples - np.mean(samples))/np.std(samples)

    def _ks_test(self):
        #Compute the KS Test Statistic and p-value
        test_stat, p_value = stats.kstest(self.samples, 'norm', args=(0, 1))
        return test_stat, p_value

    def _ad_test(self):
        #Compute the AD Test Statistic
        test_stat = stats.anderson(self.samples, dist='norm').statistic

        #Correct the test statistic for sample size
        n = len(self.samples)
        Astar = test_stat*(1 + 0.75/n + 2.25/n**2)

        #Approximate p-value from test statistic
        if Astar < 0.2:
            p_value = 1 - np.exp(-13.436 + 101.14*Astar - 223.73*Astar**2)
        elif Astar < 0.34:
            p_value = 1 - np.exp(-8.318 + 42.796*Astar - 59.938*Astar**2)
        elif Astar < 0.6:
            p_value = np.exp(0.9177 - 4.279*Astar - 1.38*Astar**2)
        elif Astar < 13:
            p_value = np.exp(1.2937 - 5.709*Astar + 0.0186*Astar**2)
        else:
            p_value = 3.7e-24 #p-value is effectively zero

        return test_stat, p_value

    def _sw_test(self):
        #Compute the SW Test Statistic and p-value
        test_stat, p_value = stats.shapiro(self.samples)
        return test_stat, p_value

    def _wwplot(self):
        #Sort the samples
        sorted_samples = np.sort(self.samples)

        #Generate the Gaussian Data
        N = len(self.samples)
        Gdata = np.sort(stats.norm.rvs(loc=0, scale=1, size=N))

        #Construct the Figure
        plt.figure(figsize=(8,8))

        #Plot the data
        plt.scatter(Gdata, sorted_samples, alpha=0.5)
        plt.plot([min(Gdata), max(Gdata)], [min(Gdata), max(Gdata)], color='red', linestyle='--')
        plt.title('QQ Plot')
        plt.xlabel('Theoretical Quantiles')
        plt.ylabel('Sample Quantiles')
        plt.grid()
        plt.axis('equal')
        plt.show()
    def test(self):
        #Generate the QQ-Plot
        self._wwplot()

        #Run all tests and return results
        ks_stat, ks_pval = self._ks_test()
        ad_stat, ad_pval = self._ad_test()
        sw_stat, sw_pval = self._sw_test()

        #Construct results dictionary
        sigN = 4 #Number of significant figures
        results = {
            'KS': {'Test Stat': f'{ks_stat:.{sigN}}', 'p-value': f'{ks_pval:.{sigN}}'},
            'AD': {'Test Stat': f'{ad_stat:.{sigN}}', 'p-value': f'{ad_pval:.{sigN}}'},
            'SW': {'Test Stat': f'{sw_stat:.{sigN}}', 'p-value': f'{sw_pval:.{sigN}}'}}

        return results



#Example Usage
N = 1000
data = stats.beta.rvs(a=2, b=5, loc=10, scale=15, size=N)
Tester = NormalityTester(data)
results = Tester.test()
display(results)
```

Output:



```
{'KS': {'Test Stat': '0.06933', 'p-value': '0.0001265'},
 'AD': {'Test Stat': '8.121', 'p-value': '8.811e-20'},
 'SW': {'Test Stat': '0.9672', 'p-value': '3.14e-14'}}
```

Notice that all of the **Normality Tests** produced **p-values** which suggest that we should reject the **Normality** of the distribution. This is expected as the distribution is not normal.

1-10 Central Limit Theorem & Residuals:

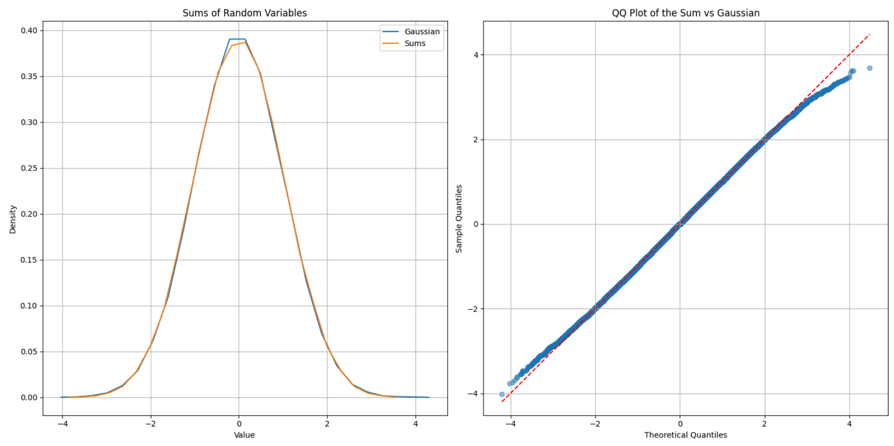
The **Central Limit Theorem** states that if you add a sufficiently large sample (n) of independent random variables with mean μ and standard deviation σ, their sum will be approximately **Normally Distributed** with mean $n\mu$ and standard deviation $\sigma\sqrt{n}$ and their average will have mean μ and standard deviation $\sigma/\sqrt{n}$. When a model is properly fit, its **Residuals** are likely to obey the **Central Limit Theorem**.

Example: Code a function which displays the distribution of the sum of num_X random variables.

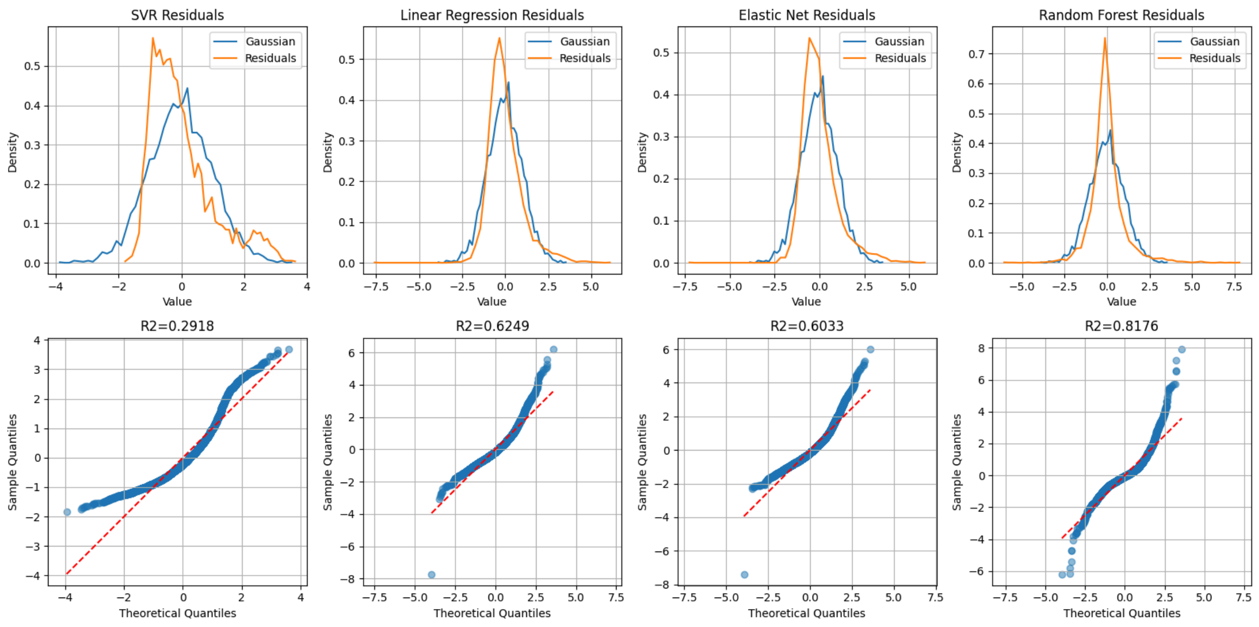
```python
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

#Define Histogram Function
def Hist_Data(data, bins=50):
    counts, bin_edges = np.histogram(data, bins=bins, density=True)
    bin_centers = 0.5*(bin_edges[1:] + bin_edges[:-1])
    return counts, bin_centers

def SumDist(N=100_000, num_X=10, bins=25):
    '''Generates num_X random variables, computes their sum, and displays the distribution of the sum'''
    #Generate Random Variables
    locs = np.random.rand(num_X)*10
    scales = 5*(np.random.rand(num_X) + 1)
    Xarr = [stats.uniform.rvs(loc=locs[i], scale=scales[i], size=N) for i in range(num_X)]

    #Compute Their Sum, Standardize
    S = np.sum(Xarr, axis=0)
    S_standardized = (S - np.mean(S))/np.std(S)

    #Compute Histogram Data
    counts_S, bins_S = Hist_Data(S_standardized, bins=bins)
    qS = np.sort(S_standardized)

    #Compute Theoretical Gaussian for Comparison
    G = stats.norm.rvs(loc=0, scale=1, size=N)
    counts_G, bins_G = Hist_Data(G, bins=bins)
    qG = np.sort(G)

    #Create the Figure
    fig, ax = plt.subplots(1, 2, figsize=(16, 8))

    #Plot the Histogram
    ax[0].plot(bins_G, counts_G, label='Gaussian')
    ax[0].plot(bins_S, counts_S, label='Sums')
    ax[0].set_title('Sums of Random Variables')
    ax[0].set_xlabel('Value')
    ax[0].set_ylabel('Density')
    ax[0].grid()
    ax[0].legend()

    #Plot the QQ Plot
    ax[1].scatter(qG, qS, alpha=0.5)
    ax[1].plot([min(qG), max(qG)], [min(qG), max(qG)], color='red', linestyle='--')
    ax[1].set_title('QQ Plot of the Sum vs Gaussian')
    ax[1].set_xlabel('Theoretical Quantiles')
    ax[1].set_ylabel('Sample Quantiles')
    ax[1].grid()
    ax[1].axis('equal')

    plt.tight_layout()
```

```
SumDist(N=100_000, num_X=10, bins=25)
```

Output:



Below is the distribution of **Residuals** for various models trained on an actual dataset (Generation code in the matching notebook):



Notice that the models with good R^2 scores also have their residuals distributed fairly normally, especially in the middle where more residuals are present, and thus, the **Central Limit Theorem** holds better. Additionally, linear models will see their estimators or **Weights** follow the same behavior.

1-11 Profiling:

For review on **Profiling** techniques, consult the updated "Profiling Examples" notebook under the "Midterm Review" module (will be posted 10/13/25 afternoon).