5.

[12:04 PM] Poornima (Guest)

# Define the functions  
def g(x):  
 return x\*\*2  
  
def h(x):  
 return 3\*x + 1  
  
# Sum function  
def sum\_func(x):  
 return g(x) + h(x)  
  
# Compute the derivative of the sum at a point, say x=1  
x\_val = 1  
sum\_derivative = derivative(sum\_func, x\_val, dx=1)  
  
print("Sum Function: g(x) + h(x) = x\*\*2 + 3\*x + 1")  
print(f"Sum Derivative at x = {x\_val}: (g + h)'(x) =", sum\_derivative)  
  
# Product function  
def product\_func(x):  
 return g(x) \* h(x)  
  
# Compute the derivative of the product at a point, say x=1  
x\_val = 1  
product\_derivative = derivative(product\_func, x\_val, dx=1)  
  
print("Product Function: g(x) \* h(x) = (x\*\*2) \* (3\*x + 1)")  
print(f"Product Derivative at x = {x\_val}: (g \* h)'(x) =", product\_derivative)  
  
# Quotient function  
def quotient\_func(x):  
 return g(x) / h(x)  
  
# Compute the derivative of the quotient at a point, say x=1  
x\_val = 1  
quotient\_derivative = derivative(quotient\_func, x\_val, dx=1)  
  
print("Quotient Function: g(x) / h(x) = (x\*\*2) / (3\*x + 1)")  
print(f"Quotient Derivative at x = {x\_val}: (g / h)'(x) =", quotient\_derivative)

[12:20 PM] Poornima (Guest)

# Define the inner and outer functions  
def g(x):  
 return np.sin(x)  
  
def f(u):  
 return np.exp(u)  
  
# Compo  
site function  
def composite\_func(x):  
 return f(g(x))  
  
# Compute the derivative using the chain rule at a point, say x=1  
x\_val = 1  
inner\_derivative = derivative(g, x\_val, dx=1)  
outer\_derivative = derivative(f, g(x\_val), dx=1)  
chain\_derivative = outer\_derivative \* inner\_derivative  
  
print("Composite Function: f(g(x)) = exp(sin(x))")  
print(f"Chain Rule Derivative at x = {x\_val}: dy/dx =", chain\_derivative)

[12:33 PM] Poornima (Guest)

# Define the function  
def f(x):  
 return np.sin(x)  
  
# Compute the first, second, and third derivatives at a point, say x=1  
x\_val = 1  
f\_prime = derivative(f, x\_val, dx=1, n=1, order=3)  
f\_double\_prime = derivative(f, x\_val, dx=1, n=2, order=5)  
f\_triple\_prime = derivative(f, x\_val, dx=1, n=3, order=7)  
  
print("Function: f(x) = sin(x)")  
print(f"First Derivative at x = {x\_val}: f'(x) =", f\_prime)  
print(f"Second Derivative at x = {x\_val}: f''(x) =", f\_double\_prime)  
print(f"Third Derivative at x = {x\_val}: f'''(x) =", f\_triple\_prime)

[12:53 PM] Poornima (Guest)

import numpy as np  
from findiff import FinDiff  
  
# Define the function  
def f(x):  
 return x\*\*2 + 3\*x + 2  
  
# Compute the derivative at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
f\_prime = d\_dx(f)(x\_val)  
  
print("Function: f(x) = x\*\*2 + 3\*x + 2")  
print(f"Derivative at x = {x\_val}: f'(x) =", f\_prime)  
  
  
  
  
# Define the functions  
def g(x):  
 return x\*\*2  
  
def h(x):  
 return 3\*x + 1  
  
# Sum function  
def sum\_func(x):  
 return g(x) + h(x)  
  
# Compute the derivative of the sum at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
sum\_derivative = d\_dx(sum\_func)(x\_val)  
  
print("Sum Function: g(x) + h(x) = x\*\*2 + 3\*x + 1")  
print(f"Sum Derivative at x = {x\_val}: (g + h)'(x) =", sum\_derivative)  
  
  
  
# Define the functions  
def g(x):  
 return x\*\*2  
  
def h(x):  
 return 3\*x + 1  
  
# Product function  
def product\_func(x):  
 return g(x) \* h(x)  
  
# Compute the derivative of the product at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
product\_derivative = d\_dx(product\_func)(x\_val)  
  
print("Product Function: g(x) \* h(x) = (x\*\*2) \* (3\*x + 1)")  
print(f"Product Derivative at x = {x\_val}: (g \* h)'(x) =", product\_derivative)  
  
  
  
# Define the functions  
def g(x):  
 return x\*\*2  
  
def h(x):  
 return 3\*x + 1  
  
# Quotient function  
def quotient\_func(x):  
 return g(x) / h(x)  
  
# Compute the derivative of the quotient at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
quotient\_derivative = d\_dx(quotient\_func)(x\_val)  
  
print("Quotient Function: g(x) / h(x) = (x\*\*2) / (3\*x + 1)")  
print(f"Quotient Derivative at x = {x\_val}: (g / h)'(x) =", quotient\_derivative)  
  
  
  
# Define the inner and outer functions  
def g(x):  
 return np.sin(x)  
  
def f(u):  
 return np.exp(u)  
  
# Composite function  
def composite\_func(x):  
 return f(g(x))  
  
# Compute the derivative using the chain rule at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
  
# Compute derivatives of inner and outer functions  
inner\_derivative = d\_dx(g)(x\_val)  
outer\_derivative = d\_dx(f)(g(x\_val))  
chain\_derivative = outer\_derivative \* inner\_derivative  
  
print("Composite Function: f(g(x)) = exp(sin(x))")  
print(f"Chain Rule Derivative at x = {x\_val}: dy/dx =", chain\_derivative)  
  
  
# Define the function  
def f(x):  
 return np.sin(x)  
  
# Compute the first, second, and third derivatives at a point, say x=1  
x\_val = 1.0  
dx = 1e-6  
d\_dx = FinDiff(0, dx, 1)  
d2\_dx2 = FinDiff(0, dx, 2)  
d3\_dx3 = FinDiff(0, dx, 3)  
  
f\_prime = d\_dx(f)(x\_val)  
f\_double\_prime = d2\_dx2(f)(x\_val)  
f\_triple\_prime = d3\_dx3(f)(x\_val)  
  
print("Function: f(x) = sin(x)")  
print(f"First Derivative at x = {x\_val}: f'(x) =", f\_prime)  
print(f"Second Derivative at x = {x\_val}: f''(x) =", f\_double\_prime)  
print(f"Third Derivative at x = {x\_val}: f(triple dash)(x) = ", f\_triple\_prime)

Applications

[2:54 PM] Poornima (Guest)

import numpy as np

# linear transformation matrix

A = np.array([

[2, 0],

[0, 3]

])

# vector

x = np.array([1, 2])

y = np.dot(A, x)

print("Matrix A:\n", A)

print("Vector x:", x)

print("Transformed vector y:", y)

#========================================================================

import numpy as np

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# Generate data

np.random.seed(0)

data = np.random.rand(5, 2)

print('\n\nPCA Data\n', data)

# Standardize the data

data\_mean = np.mean(data, axis=0)

data\_centered = data - data\_mean

# Apply PCA

pca = PCA(n\_components=2)

principal\_components = pca.fit\_transform(data\_centered)

print("Principal Components:\n", principal\_components)

# Plot the data and the principal components

plt.scatter(principal\_components[:, 0], principal\_components[:, 1])

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.title('PCA of the Data')

plt.show()

#======================================================

import numpy as np

A = np.array([

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

])

# Compute the SVD

U, S, Vt = np.linalg.svd(A)

print('\n\nSVD')

print("Matrix A:\n", A)

print("Matrix U:\n", U)

print("Singular Values:", S)

print("Matrix V^T:\n", Vt)

# Reconstruct the matrix using SVD components

Sigma = np.zeros((A.shape[0], A.shape[1]))

Sigma[:A.shape[1], :A.shape[1]] = np.diag(S)

A\_reconstructed = np.dot(U, np.dot(Sigma, Vt))

print("Reconstructed Matrix A:\n", A\_reconstructed)

#==========================================================================

Partial derivative

[3:17 PM] Poornima (Guest)

import sympy as sp  
  
x, y = sp.symbols('x y')  
f = x\*\*2 + y\*\*2  
  
# Compute the partial derivatives  
f\_x = sp.diff(f, x)  
f\_y = sp.diff(f, y)  
  
print(f"Partial derivative with respect to x: {f\_x}")  
print(f"Partial derivative with respect to y: {f\_y}")  
  
# Compute the gradient  
gradient\_f = (f\_x, f\_y)  
print(f"Gradient of f: {gradient\_f}")  
  
#=======================================================================  
  
# Define the variables and the vector-valued function  
x, y = sp.symbols('X Y')  
F = sp.Matrix([x\*\*2 + y\*\*2, sp.sin(x) + sp.cos(y)])  
print('F: ', F)  
print(x,y)  
  
# Compute the Jacobian matrix  
J = F.jacobian([x, y])  
print("Jacobian matrix:")  
sp.pprint(J)  
  
  
x, y = sp.symbols('x y')  
f = x\*\*2 + y\*\*2  
print(x,y)  
print('f: ',f)  
# Compute the Hessian matrix  
H = sp.hessian(f, (x, y))  
print("Hessian matrix:")  
sp.pprint(H)  
  
#=====================================================================

7.integrals

[5:01 PM] Poornima (Guest)

import sympy as sp  
  
# Define the variable and the function  
x = sp.symbols('x')  
f = x\*\*2 + 3\*x + 2  
  
# Compute the indefinite integral  
F = sp.integrate(f, x)  
  
print("Indefinite integral of f(x) = x\*\*2 + 3\*x + 2:")  
sp.pprint(F)  
  
#=============================================================  
  
import sympy as sp  
  
# Define the variable and the function  
x = sp.symbols('x')  
f = x\*\*2 + 3\*x + 2  
  
# Compute the definite integral from 1 to 3  
a, b = 1, 3  
def\_integral = sp.integrate(f, (x, a, b))  
print(f"Definite integral of f(x) = x\*\*2 + 3\*x + 2 from {a} to {b}:")  
sp.pprint(def\_integral)  
  
#==========================================================================  
  
import sympy as sp  
  
# Define the variables and the functions  
x = sp.symbols('x')  
f = x\*\*2  
g = 3\*x  
h = x\*\*2 + 3\*x  
  
# Power rule  
int\_power = sp.integrate(x\*\*2, x)  
print("Integral of x^2:")  
sp.pprint(int\_power)  
  
# Constant multiple rule  
int\_const\_multiple = sp.integrate(3\*x, x)  
print("Integral of 3x:")  
sp.pprint(int\_const\_multiple)  
  
# Sum rule  
int\_sum = sp.integrate(h, x)  
print("Integral of (x^2 + 3x):")  
sp.pprint(int\_sum)  
  
  
#+========================================================================

8.statical

[5:45 PM] Poornima (Guest)

import numpy as np  
from scipy import stats  
  
data = [1, 5, 6, 2, 3, 4]  
 # 1 3 4 5 6 7 9  
data = np.array(data)  
  
mean = np.mean(data)  
print(f"Mean: {mean}")  
median = np.median(data)  
print(f"Median: {median}")  
mode\_result = stats.mode(data)  
print(mode\_result)  
mode = mode\_result[0] if len(mode\_result) > 0 else None  
count = mode\_result[1] if len(mode\_result) > 0 else None  
print(f"Mode: {mode} Count: {count}")  
  
variance = np.var(data, ddof=1) # ddof=1 for sample variance  
print(f"Variance: {variance}")  
  
std\_dev = np.std(data, ddof=1)  
print(f"Standard Deviation: {std\_dev}")  
  
data\_range = np.ptp(data)  
print(f"Range: {data\_range}")  
  
skewness = stats.skew(data)  
print(f"Skewness: {skewness}")  
  
kurtosis = stats.kurtosis(data)  
print(f"Kurtosis: {kurtosis}")  
  
#=====================================================================  
  
import matplotlib.pyplot as plt  
  
# Visualization Techniques  
# Histogram  
plt.hist(data, bins=5, edgecolor='black')  
plt.title('Histogram')  
plt.xlabel('Value')  
plt.ylabel('Frequency')  
plt.show()  
  
# Box Plot  
plt.boxplot(data)  
plt.title('Box Plot')  
plt.ylabel('Value')  
plt.show()  
  
#==========================================================================

9.probability

[9:44 AM] Poornima (Guest)

import numpy as np  
  
# Sample Space and Events  
sample\_space = {'Heads', 'Tails'}  
event = {'Heads'}  
  
print(f"Sample Space: {sample\_space}")  
print(f"Event: {event}")  
#=============  
sample\_space = ['H', 'T']  
probabilities = [0.5, 0.5]  
  
# Complement Rule  
P\_H = 0.5  
P\_not\_H = 1 - P\_H  
print(f"P(H): {P\_H}")  
print(f"P(not H): {P\_not\_H}")  
  
# Addition Rule  
P\_A = 0.4  
P\_B = 0.3  
P\_A\_and\_B = 0.1  
P\_A\_or\_B = P\_A + P\_B - P\_A\_and\_B  
print(f"P(A or B): {P\_A\_or\_B}")  
  
# Multiplication Rule  
P\_A\_given\_B = 0.5  
P\_B = 0.3  
P\_A\_and\_B = P\_B \* P\_A\_given\_B  
print(f"P(A and B): {P\_A\_and\_B}")  
  
# Conditional Probability  
P\_E\_and\_F = 0.1  
P\_F = 0.3  
P\_E\_given\_F = P\_E\_and\_F / P\_F  
print(f"P(E|F): {P\_E\_given\_F}")  
  
# Bayes' Theorem  
P\_F\_given\_E = 0.5  
P\_E = 0.4  
P\_F = 0.3  
P\_E\_given\_F = (P\_F\_given\_E \* P\_E) / P\_F  
print(f"P(E|F) using Bayes' Theorem: {P\_E\_given\_F}")  
  
# Independence Check  
P\_E = 0.4  
P\_F = 0.3  
P\_E\_and\_F = 0.12  
are\_independent = np.isclose(P\_E\_and\_F, P\_E \* P\_F)  
print(f"Are E and F independent? {are\_independent}")

10.probability distributions

[10:39 AM] Poornima (Guest)

import numpy as np  
from scipy.stats import bernoulli, binom, poisson  
  
#DISCRETE DISTRIBUTION  
# Bernoulli Distribution  
p = 0.5  
bernoulli\_rv = bernoulli(p)  
print(f"Bernoulli PMF: P(X=1) = {bernoulli\_rv.pmf(1)}, P(X=0) = {bernoulli\_rv.pmf(0)}")  
  
# Binomial Distribution  
n = 10  
p = 0.5  
binom\_rv = binom(n, p)  
k = 5  
print(f"Binomial PMF: P(X={k}) = {binom\_rv.pmf(k)}")  
  
# Poisson Distribution  
lam = 3  
poisson\_rv = poisson(lam)  
k = 4  
print(f"Poisson PMF: P(X={k}) = {poisson\_rv.pmf(k)}")  
  
  
#CONTINUOUS DISTRIBUTION  
from scipy.stats import uniform, norm, expon  
  
# Uniform Distribution  
a, b = 0, 1  
uniform\_rv = uniform(a, b-a)  
print(f"Uniform PDF: f(x=0.5) = {uniform\_rv.pdf(0.5)}")  
  
# Normal Distribution  
mu, sigma = 0, 1  
norm\_rv = norm(mu, sigma)  
print(f"Normal PDF: f(x=0) = {norm\_rv.pdf(0)}")

[12:00 PM] Poornima (Guest)

#PROP & VAL  
from scipy.stats import norm  
  
# Normal Distribution  
mu, sigma = 0, 1  
norm\_rv = norm(mu, sigma)  
  
# Mean  
mean = norm\_rv.mean()  
print(f"Mean: {mean}")  
  
# Variance  
variance = norm\_rv.var()  
print(f"Variance: {variance}")  
  
# Skewness  
skewness = norm\_rv.stats(moments='s')  
print(f"Skewness: {skewness}")  
  
# Kurtosis  
kurtosis = norm\_rv.stats(moments='k')  
print(f"Kurtosis: {kurtosis}")

11.inferential statistics

[12:55 PM] Poornima (Guest)

import numpy as np  
  
# Population data  
population = np.random.normal(loc=50, scale=10, size=10000)  
  
# Simple Random Sampling  
sample\_size = 100  
sample = np.random.choice(population, sample\_size)  
  
# Sample mean and standard deviation  
sample\_mean = np.mean(sample)  
sample\_std = np.std(sample, ddof=1)  
  
print(f"Sample Mean: {sample\_mean}")  
print(f"Sample Standard Deviation: {sample\_std}")  
  
# Sampling Distribution of the Sample Mean  
num\_samples = 1000  
sample\_means = [np.mean(np.random.choice(population, sample\_size))  
 for \_ in range(num\_samples)]  
  
# Mean and standard deviation of the sampling distribution  
sampling\_mean = np.mean(sample\_means)  
sampling\_std = np.std(sample\_means)  
  
print(f"Sampling Distribution Mean: {sampling\_mean}")  
print(f"Sampling Distribution Standard Deviation: {sampling\_std}")  
  
#===========================================================  
  
import scipy.stats as stats  
  
# Sample data  
data = np.random.normal(loc=50, scale=10, size=100)  
  
# Point estimate (sample mean)  
point\_estimate = np.mean(data)  
  
# Confidence interval  
confidence\_level = 0.95  
degrees\_freedom = len(data) - 1  
sample\_mean = np.mean(data)  
sample\_std = np.std(data, ddof=1)  
confidence\_interval = stats.t.interval(confidence\_level, degrees\_freedom, sample\_mean, sample\_std/np.sqrt(len(data)))  
  
print(f"Point Estimate (Sample Mean): {point\_estimate}")  
print(f"{confidence\_level\*100}% Confidence Interval: {confidence\_interval}")  
  
#============================================================================  
  
import scipy.stats as stats  
  
# Sample data  
data = np.random.normal(loc=50, scale=10, size=100)  
  
# Hypothesis test  
# Null hypothesis: mean = 50  
# Alternative hypothesis: mean != 50  
null\_hypothesis\_mean = 50  
t\_statistic, p\_value = stats.ttest\_1samp(data, null\_hypothesis\_mean)  
  
print(f"T-statistic: {t\_statistic}")  
print(f"P-value: {p\_value}")  
  
# Decision  
alpha = 0.05  
if p\_value < alpha:  
 print("Reject the null hypothesis")  
else:  
 print("Fail to reject the null hypothesis")  
  
  
#===============================================================  
  
  
import numpy as np  
import scipy.stats as stats  
  
# Sample data  
data = np.random.normal(loc=50, scale=10, size=100)  
  
# Confidence interval for the mean  
confidence\_level = 0.95  
degrees\_freedom = len(data) - 1  
sample\_mean = np.mean(data)  
sample\_std = np.std(data, ddof=1)  
confidence\_interval = stats.t.interval(confidence\_level, degrees\_freedom, sample\_mean, sample\_std/np.sqrt(len(data)))  
  
print(f"{confidence\_level\*100}% Confidence Interval: {confidence\_interval}")  
  
# Significance level and p-value  
# Null hypothesis: mean = 50  
null\_hypothesis\_mean = 50  
t\_statistic, p\_value = stats.ttest\_1samp(data, null\_hypothesis\_mean)  
  
print(f"Significance Level (alpha): 0.05")  
print(f"P-value: {p\_value}")  
  
if p\_value < 0.05:  
 print("Reject the null hypothesis")  
else:  
 print("Fail to reject the null hypothesis")  
  
#==========================================================

statistical\_testing

1.hypothesis\_testing

[3:50 PM] Poornima (Guest)

from scipy import stats  
  
'''  
1-SAMPLE T-TEST  
  
Formulating Hypotheses:  
• H0 : μ = 300  
• H1 : μ ≠ 300  
  
The t-test checks whether the sample mean is significantly   
different from the population mean.   
Since the sample mean is very close to 300, the p-value will be higher,  
indicating insufficient evidence to reject the null hypothesis   
at the 0.05 significance level.  
'''  
  
# lifespans of new batteries (in hours)  
#data = [310, 320, 290, 330, 340, 300, 310, 320, 305, 325]  
  
data = [295, 302, 298, 301, 299, 300, 297, 303, 296, 300]  
  
# Population mean of old batteries  
population\_mean = 300  
# Perform one-sample t-test  
t\_statistic, p\_value = stats.ttest\_1samp(data, population\_mean)  
alpha = 0.05  
print(f"T-statistic: {t\_statistic}")  
print(f"P-value: {p\_value}")  
if p\_value < alpha:  
 print("Reject the null hypothesis \n The new battery lasts longer than the old battery.")  
else:  
 print("Fail to reject the null hypothesis \n There is no significant difference in battery life.")

[5:17 PM] Poornima (Guest)

'''  
1-SAMPLE T-TEST  
  
Examine whether a new teaching method affects the average test scores of students.   
We hypothesize that the new teaching method has not changed the average test score  
from the traditional method, which is known to be 75.  
   
One-Sample t-Test for Average Test Scores  
Formulating Hypotheses  
• Null Hypothesis (H0): The mean test score with the new teaching method is equal to 75.  
H0 : μ <= 75  
• Alternative Hypothesis (H1): The mean test score with the new teaching method is different from 75.  
H1 : μ > 75  
  
'''  
  
from scipy import stats  
  
# Sample data: test scores of students taught using the new method  
test\_scores = [78, 74, 82, 70, 75, 77, 79, 81, 73, 76]  
  
population\_mean = 75  
alpha = 0.05  
  
t\_statistic, p\_value = stats.ttest\_1samp(test\_scores, population\_mean)  
  
print(f"T-statistic: {t\_statistic}")  
print(f"P-value: {p\_value}")  
  
if p\_value < alpha:  
 print("Reject the null hypothesis: The new teaching method has changed the average test score.")  
else:  
 print("Fail to reject the null hypothesis: There is no significant difference in average test scores.")  
  
#=====================================================================  
  
'''  
2-SAMPLE T-TEST ==> Independent   
H0 : There is no significant change between the 2 methods   
There is no significant difference between the two teaching methods  
H1 : There is significant change between the 2 methods and method 2 is better  
There is a significant difference between the two teaching methods.  
'''  
  
from scipy import stats  
  
# Sample data: test scores from two different teaching methods  
method\_1\_scores = [78, 74, 82, 70, 75, 77, 79, 81, 73, 76]  
method\_2\_scores = [80, 85, 78, 82, 84, 79, 81, 83, 80, 82]  
  
# Perform independent samples t-test  
t\_statistic, p\_value = stats.ttest\_ind(method\_1\_scores, method\_2\_scores)  
  
alpha = 0.05  
  
print(f"T-statistic: {t\_statistic}")  
print(f"P-value: {p\_value}")  
if p\_value < alpha:  
 print("Reject the null hypothesis: There is a significant difference between the two teaching methods.")  
else:  
 print("Fail to reject the null hypothesis: There is no significant difference between the two teaching methods.")  
  
  
#=======================================================================================  
  
  
  
from scipy import stats  
  
  
'''  
2-SAMPLE T-TEST ==> Dependent / Paired  
  
H0: There is no significant difference between the test scores  
 before and after the new teaching method.  
H1: There is a significant difference between the test scores   
before and after the new teaching method.  
'''  
# Sample data: test scores before and after using a new teaching method  
before\_scores = [78, 74, 82, 70, 75, 77, 79, 81, 73, 76]  
after\_scores = [80, 85, 78, 82, 84, 79, 81, 83, 80, 82]  
  
# Perform paired samples t-test  
t\_statistic, p\_value = stats.ttest\_rel(before\_scores, after\_scores)  
  
alpha = 0.05  
  
print(f"T-statistic: {t\_statistic}")  
print(f"P-value: {p\_value}")  
  
if p\_value < alpha:  
 print("Reject the null hypothesis: There is a significant difference between the test scores before and after the new teaching method.")  
else:  
 print("Fail to reject the null hypothesis: There is no significant difference between the test scores before and after the new teaching method.")  
  
#===================================================

2.chi\_squared

[10:21 AM] Poornima (Guest)

#GOODNESS FIT  
import numpy as np  
from scipy import stats  
  
# Observed frequencies  
observed = np.array([8, 12, 11, 9, 10, 10]) #60 times  
#observed = np.array([18, 5, 10, 12, 10, 5]) #60 times  
  
# Expected frequencies (assuming a fair die)  
expected = np.array([10, 10, 10, 10, 10, 10]) #60 times  
  
# Perform the Chi-Squared Goodness of Fit test  
chi\_squared\_stat, p\_value = stats.chisquare(observed, f\_exp=expected)  
  
# Degrees of freedom  
degrees\_of\_freedom = len(observed) - 1  
  
# Significance level (alpha)  
alpha = 0.05  
critical\_value = 10  
  
print(f"Chi-Squared Statistic: {chi\_squared\_stat}")  
print(f"P-value: {p\_value}")  
print(f"Degrees of Freedom: {degrees\_of\_freedom}")  
  
# Decision  
if p\_value < alpha:  
 print("Reject the null hypothesis: The observed frequencies do not match the expected frequencies.")  
else:  
 print("Fail to reject the null hypothesis: The observed frequencies match the expected frequencies.")  
  
if chi\_squared\_stat > critical\_value:  
 print('Observed freq may not be a good fit')  
  
  
#==============================================================  
  
#TEST OF INDEPENDENCE  
import numpy as np  
from scipy import stats  
  
# Create a contingency table  
# Rows: Age Group (Under 30, 30-50, Over 50)  
# Columns: Product Preference (Preferred, Not Preferred)  
observed = np.array([[30, 10], [35, 15], [25, 20]])  
  
# Perform the Chi-Squared Test of Independence  
chi\_squared\_stat, p\_value, dof, expected = stats.chi2\_contingency(observed)  
  
print(f"Chi-Squared Statistic: {chi\_squared\_stat}")  
print(f"P-value: {p\_value}")  
print(f"Degrees of Freedom: {dof}")  
print(f"Expected Frequencies:\n{expected}")  
  
# Significance level (alpha)  
alpha = 0.05  
  
# Decision  
if p\_value < alpha:  
 print("Reject the null hypothesis: There is an association between age group and product preference.")  
else:  
 print("Fail to reject the null hypothesis: There is no association between age group and product preference.")  
  
#=========================================================  
  
  
  
  
  
  
  
3.anova

[12:08 PM] Poornima (Guest)

import pandas as pd  
from scipy import stats  
  
'''  
  
Test scores from three different teaching methods.  
One-way ANOVA test  
'''  
  
'''data = {  
 'Method A': [85, 87, 88, 94, 78],  
 'Method B': [80, 85, 84, 89, 82],  
 'Method C': [78, 82, 83, 88, 90]  
}  
'''  
  
data = {  
 'Method A': [55, 67, 58, 64, 67],  
 'Method B': [80, 85, 84, 89, 82],  
 'Method C': [78, 82, 83, 88, 90]  
}  
  
# Convert data to a DataFrame  
df = pd.DataFrame(data)  
print(df)  
  
# Perform one-way ANOVA  
f\_statistic, p\_value = stats.f\_oneway(df['Method A'],  
 df['Method B'], df['Method C'])  
  
print(f"F-statistic: {f\_statistic}")  
print(f"P-value: {p\_value}")  
  
# Significance level  
alpha = 0.05  
  
# Decision  
if p\_value < alpha:  
 print("Reject the null hypothesis: At least one teaching method is significantly different.")  
else:  
 print("Fail to reject the null hypothesis: No significant difference between teaching methods.")  
  
#==================================================================================  
  
import pandas as pd  
from scipy import stats  
  
# Sample data: Heights of plants (in cm) under 4 different fertilizers  
data = {  
 'Fertilizer A': [22, 24, 21, 23, 26],  
 'Fertilizer B': [25, 27, 28, 24, 23],  
 'Fertilizer C': [20, 21, 19, 22, 18],  
 'Fertilizer D': [27, 21, 21, 22, 28]  
}  
  
# Convert data to a DataFrame  
df = pd.DataFrame(data)  
print(df)  
  
# Perform one-way ANOVA  
f\_statistic, p\_value = stats.f\_oneway(df['Fertilizer A'],  
 df['Fertilizer B'], df['Fertilizer C'],  
 df['Fertilizer D'])  
  
print(f"F-statistic: {f\_statistic}")  
print(f"P-value: {p\_value}")  
  
# Significance level  
alpha = 0.05  
  
# Decision  
if p\_value < alpha:  
 print("Reject the null hypothesis: At least one fertilizer is significantly different.")  
else:  
 print("Fail to reject the null hypothesis: No significant difference between fertilizers.")  
  
#===============================================================================  
  
import numpy as np  
import pandas as pd  
from scipy import stats  
from statsmodels.stats.multicomp import pairwise\_tukeyhsd  
  
# Sample data: Heights of plants (in cm) under three different fertilizers  
data = {  
 'Fertilizer': np.repeat(['A', 'B', 'C'], 5),  
 'Height': [22, 24, 21, 23, 26, 25, 27, 28, 24, 23, 20, 21, 19, 22, 18]  
}  
  
# Convert data to a DataFrame  
df = pd.DataFrame(data)  
print(df)  
  
# Perform Tukey's HSD post-hoc test  
tukey\_result = pairwise\_tukeyhsd(endog=df['Height'],  
 groups=df['Fertilizer'], alpha=0.05)  
  
print(tukey\_result)  
  
# Plot the results  
tukey\_result.plot\_simultaneous()  
  
#============================================================

numpy

1 numpy basics

[2:38 PM] Poornima (Guest)

'''  
NumPy Arrays  
Creating Arrays from Lists  
'''  
  
import numpy as np  
  
# Creating a 1D array from a list  
list\_1d = [1, 2, 3, 4, 5]  
array\_1d = np.array(list\_1d)  
print("1D Array from list:", array\_1d)  
  
# Creating a 2D array from a list of lists  
list\_2d = [[1, 2, 3], [4, 5, 6]]  
array\_2d = np.array(list\_2d)  
print("2D Array from list of lists:\n", array\_2d)  
  
  
'''  
Array Attributes  
Shape and Size: Get the shape and size of a NumPy array using the shape  
 and size attributes.  
Data Types: NumPy arrays have a single data type for all elements.   
You can check the data type with the dtype attribute.  
Specify the data type when creating an array.  
  
'''  
  
array = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Shape of the array  
print("Shape of array:", array.shape)  
  
# Size of the array  
print("Size of array:", array.size)  
  
# Data type of the array  
print("Data type of array:", array.dtype)  
  
array\_float = np.array([1, 2, 3], dtype=np.float64)  
print("Array with specified data type:", array\_float)  
print("Data type of the array:", array\_float.dtype)

'''  
Built-in Functions  
arange: Similar to Python's range() but returns a NumPy array.  
linspace: Creates an array of evenly spaced values over a specified range.  
ones: Creates an array filled with ones.  
zeros: Creates an array filled with zeros.  
'''  
  
array\_arange = np.arange(0, 10, 2)  
print("Array with arange:", array\_arange)  
  
array\_linspace = np.linspace(10, 100, 5)  
print("Array with linspace:", array\_linspace)  
  
array\_ones = np.ones((2, 3))  
print("Array of ones:\n", array\_ones)  
  
array\_zeros = np.zeros((2, 3))  
print("Array of zeros:\n", array\_zeros)

'''  
Random Arrays  
rand: Creates an array of given shape with random values between 0 and 1.  
randn: Creates an array of given shape with random values from a   
standard normal distribution.  
randint: Creates an array with random integers within a specified range.  
  
'''  
  
array\_rand = np.random.rand(2, 3)  
print("Random array with rand:\n", array\_rand)  
  
array\_randn = np.random.randn(2, 3)  
print("Random array with randn:\n", array\_randn)  
  
array\_randint = np.random.randint(0, 10, (2, 3))  
print("Random array with randint:\n", array\_randint)  
  
  
'''  
Indexing and Slicing  
Reshaping Arrays  
Change the shape of an array without changing its data using the  
reshape() method.  
  
'''  
  
array = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Indexing  
print("Element at [0, 1]:", array[0, 1])  
  
# Slicing  
print("First row:", array[0, :])  
print("First column:", array[:, 0])  
print("Sub-array:", array[0:2, 1:3])  
  
array = np.arange(6)  
print("Original array:", array)  
  
reshaped\_array = array.reshape((2, 3))  
print("Reshaped array:\n", reshaped\_array)  
  
  
#===================================================================

2.array operations

[3:39 PM] Poornima (Guest)

'''  
Array Operations  
Element-wise Operations : NumPy supports element-wise operations,  
which apply operations to each element in the array individually.  
  
'''  
  
import numpy as np  
  
array1 = np.array([1, 2, 3])  
array2 = np.array([4, 5, 6])  
  
# Element-wise addition  
print("Element-wise addition:", array1 + array2)  
  
# Element-wise subtraction  
print("Element-wise subtraction:", array1 - array2)  
  
# Element-wise multiplication  
print("Element-wise multiplication:", array1 \* array2)  
  
# Element-wise division  
print("Element-wise division:", array1 / array2)  
  
'''  
Basic Arithmetic Operations  
NumPy provides functions for basic arithmetic operations which also operate element-wise.  
  
Aggregate Functions  
NumPy provides aggregate functions that operate over the entire array   
or along a specific axis.  
  
'''  
  
# Adding a scalar to an array  
print("Adding 10 to each element:", array1 + 10)  
  
# Multiplying each element by a scalar  
print("Multiplying each element by 2:", array1 \* 2)  
  
# Using numpy functions  
print("Square of each element:", np.square(array1))  
print("Square root of each element:", np.sqrt(array1))  
  
  
array = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Sum of all elements  
print("Sum of all elements:", np.sum(array))  
  
# Mean of all elements  
print("Mean of all elements:", np.mean(array))  
  
# Minimum element  
print("Minimum element:", np.min(array))  
  
# Maximum element  
print("Maximum element:", np.max(array))  
  
# Sum along each column  
print("Sum along each column:", np.sum(array, axis=0))  
  
# Sum along each row  
print("Sum along each row:", np.sum(array, axis=1))  
  
'''  
Advanced Array Operations  
Broadcasting  
Broadcasting allows NumPy to perform operations on arrays   
of different shapes. The smaller array is "broadcast" to the shape   
of the larger array.  
  
Vectorized Operations  
Vectorized operations in NumPy allow you to perform batch operations  
 on data without writing explicit loops, leading to more concise   
 and faster code.  
  
Array Sorting and Searching  
NumPy provides efficient functions for sorting and searching arrays.  
  
'''  
  
array1 = np.array([1, 2, 3])  
array2 = np.array([[1], [2], [3]])  
  
# Broadcasting array1 to match the shape of array2  
print("Broadcasting array1 to match array2:\n", array1 + array2)  
  
# Broadcasting scalar to array  
print("Adding scalar 10 to array:\n", array1 + 10)  
  
  
  
array = np.arange(1, 6)  
  
# Vectorized operation: Adding 10 to each element  
vectorized\_addition = array + 10  
print("Vectorized addition:", vectorized\_addition)  
  
# Vectorized operation: Squaring each element  
vectorized\_square = np.square(array)  
print("Vectorized squaring:", vectorized\_square)  
  
  
array = np.array([3, 1, 4, 1, 5, 9])  
  
# Sorting an array  
sorted\_array = np.sort(array)  
print("Sorted array:", sorted\_array)  
  
# Finding indices of sorted elements  
sorted\_indices = np.argsort(array)  
print("Indices of sorted elements:", sorted\_indices)  
  
# Searching for elements  
# Finding index of first occurrence of 1  
index\_of\_one = np.where(array == 1)  
print("Indices where element is 1:", index\_of\_one)  
  
# Checking if any element is greater than 5  
any\_greater\_than\_five = np.any(array > 5)  
print("Any element greater than 5:", any\_greater\_than\_five)  
  
# Checking if all elements are positive  
all\_positive = np.all(array > 0)  
print("All elements are positive:", all\_positive)  
  
#===============================================================

3.mathematical\_fns.py

[5:43 PM] Poornima (Guest)

'''  
Basic Mathematical Functions  
Trigonometric Functions  
NumPy provides various trigonometric functions such as sine, cosine,  
tangent, etc.  
'''  
  
import numpy as np  
  
angles = np.array([0, np.pi/2, np.pi])  
print(angles)  
print("Sine of angles:", np.sin(angles))  
print("Cosine of angles:", np.cos(angles))  
print("Tangent of angles:", np.tan(angles))  
  
# Inverse trigonometric functions  
print("Arcsine of 1:", np.arcsin(1))  
print("Arccosine of 0:", np.arccos(0))  
print("Arctangent of 1:", np.arctan(1))  
  
'''  
Exponential and Logarithmic Functions  
Rounding and Modulus Functions  
'''  
  
values = np.array([1, 2, 3])  
  
print("Exponential of values:", np.exp(values))  
print("Natural log of values:", np.log(values))  
print("Base-10 log of values:", np.log10(values))  
  
values = np.array([1.7, 2.3, 3.9])  
  
print("Floor of values:", np.floor(values))  
print("Ceil of values:", np.ceil(values))  
print("Rounded values:", np.round(values))  
  
print("Modulus of 5 and 2:", np.mod(7, -3))  
print("Remainder of 5 divided by 2:", np.remainder(7, -3))  
print(7%-3)  
  
'''  
Linear Algebra Functions  
Dot Product and Matrix Multiplication  
Determinants and Inverses  
Eigenvalues and Eigenvectors  
  
'''  
  
A = np.array([[1, 2], [3, 4]])  
B = np.array([[5, 6], [7, 8]])  
  
print("Dot product of A and B:", np.dot(A, B))  
print("Matrix multiplication of A and B:", np.matmul(A, B))  
  
print("Determinant of A:", np.linalg.det(A))  
print("Inverse of A:\n", np.linalg.inv(A))  
  
eigenvalues, eigenvectors = np.linalg.eig(A)  
print("Eigenvalues of A:", eigenvalues)  
print("Eigenvectors of A:\n", eigenvectors)  
  
'''  
Statistical Functions  
Mean, Median, and Mode  
Variance and Standard Deviation  
  
'''  
  
from scipy import stats  
  
data = np.array([1, 2, 2, 3, 4])  
  
print("Mean of data:", np.mean(data))  
print("Median of data:", np.median(data))  
mode\_result = stats.mode(data)  
mode = mode\_result[0] if mode\_result.mode.size > 0 else None  
print("Mode of data:", mode)  
  
  
print("Variance of data:", np.var(data))  
print("Standard deviation of data:", np.std(data))  
  
'''  
Correlation and Covariance  
  
'''  
  
data1 = np.array([1, 2, 3, 4, 5])  
data2 = np.array([5, 4, 3, 2, 1])  
  
print("Correlation coefficient between data1 and data2:\n", np.corrcoef(data1, data2))  
print("Covariance matrix of data1 and data2:\n", np.cov(data1, data2))  
  
'''  
Random Number Generation  
Generating Random Numbers  
Setting Random Seeds  
Random Sampling from Distributions  
  
'''  
  
print("Random numbers from uniform distribution:", np.random.rand(5))  
print("Random numbers from normal distribution:", np.random.randn(5))  
print("Random integers between 1 and 10:", np.random.randint(1, 10, size=5))  
  
np.random.seed(42)  
print("Random numbers with seed 42:", np.random.rand(10))  
  
np.random.seed(22)  
print("Reproducible random numbers with seed 42:", np.random.rand(5))  
  
  
# Normal distribution  
samples = np.random.normal(loc=0, scale=1, size=10)  
print("Samples from normal distribution:", samples)  
  
# Uniform distribution  
samples = np.random.uniform(low=0, high=1, size=10)  
print("Samples from uniform distribution:", samples)  
  
#======================================================================