**ML LAB INTERNALS**

**1..([115.3, 195.5, 120.5, 110.2, 90.4, 105.6, 110.9, 116.3, 122.3, 125.4])**

**Use the above array of values and compute the mean, median, mode, standard deviation, variance** min**-max normalization and standardization.**

A)

import numpy as np

from scipy import stats

# Given array

data = [115.3, 195.5, 120.5, 110.2, 90.4, 105.6, 110.9, 116.3, 122.3, 125.4]

# Mean

mean\_value = np.mean(data)

print(f"Mean: {mean\_value}")

# Median

median\_value = np.median(data)

print(f"Median: {median\_value}")

# Mode

mode\_value = stats.mode(data)[0][0]

print(f"Mode: {mode\_value}")

# Standard Deviation

std\_dev = np.std(data)

print(f"Standard Deviation: {std\_dev}")

# Variance

variance = np.var(data)

print(f"Variance: {variance}")

# Min-Max Normalization

min\_max\_normalized = (data - np.min(data)) / (np.max(data) - np.min(data))

print(f"Min-Max Normalized: {min\_max\_normalized}")

# Standardization (Z-score)

z\_score\_normalized = (data - np.mean(data)) / np.std(data)

print(f"Standardized (Z-score): {z\_score\_normalized}")

B)

import statistics as st

arr = [115.3, 195.5, 120.5,120.5,120.5, 110.2, 90.4, 105.6, 110.9, 116.3, 122.3, 125.4]

sum = 0

for num in arr:

sum += num

mean = sum/(len(arr))

print("mean = ", mean)

arr1 = sorted(arr)

median = (arr1[len(arr1) // 2 - 1] + arr1[len(arr1) // 2]) / 2

print("median = ", median)

mode = {}

for ele in arr:

if ele not in mode:

mode[ele] = 0

else:

mode[ele] += 1

count = [g for g,l in mode.items() if l==max(mode.values())]

print("mode = ",count[0])

print("sd = ",st.stdev(arr))

print("varience = ",st.variance(arr))

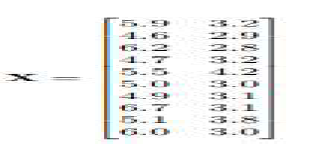
print("max = ", max(arr))

print("min = ", min(arr))

**2. K-Means Clustering is an Unsupervised Machine Learning algorithm, which groups the unlabelled dataset into different clusters . The goal of clustering is to divide the population or set of data points into a number of groups so that the data points within each group are more comparable to one another and different from the data points within the other groups. It is essentially a grouping of things based on how similar and different they are to one another.**

**Given the matrix X whose rows represent different data points, run k-means clustering on this dataset using the Euclidean distance as the distance function. Here k is chosen as 3. The centres of 3 clusters were initialized as μ1 = (6.2, 3.2) (red), μ2 = (6.6, 3.7) (green), μ3 = (6.5, 3.0) (blue).**

**What's the centre of the second cluster (green) after two iteration? What's the centre of the third cluster (blue) when the clustering converges? How many iterations are required for the clusters to converge?**

****

import numpy as np

from sklearn.cluster import KMeans

# Given data

X = np.array([[5.9,3.2],

[4.6,2.9],

[6.2,2.8],

[4.7,3.2],

[5.5,4.2],

[5.0,3.0],

[4.9,3.1],

[6.7,3.1],

[5.1,3.8],

[6.0,3.0]

])

# Initialize cluster centers

initial\_centers = np.array([[6.2, 3.2], # μ1 (red)

[6.6, 3.7], # μ2 (green)

[6.5, 3.0] # μ3 (blue)

])

# Run k-means clustering with k=3

kmeans = KMeans(n\_clusters=3, init=initial\_centers, n\_init=1, algorithm='full', random\_state=42)

kmeans.fit(X)

# Get cluster centers after one iteration

centers\_iteration\_1 = kmeans.cluster\_centers\_

# Get cluster centers after two iterations

kmeans.fit(X) # Fit again to perform the second iteration

centers\_iteration\_2 = kmeans.cluster\_centers\_

# Get cluster centers when clustering converges

converged\_centers = kmeans.cluster\_centers\_

# Get the number of iterations required for convergence

num\_iterations\_to\_converge = kmeans.n\_iter\_

# Round the results to three decimal places

centers\_iteration\_1 = np.round(centers\_iteration\_1, 3)

centers\_iteration\_2 = np.round(centers\_iteration\_2, 3)

converged\_centers = np.round(converged\_centers, 3)

print(f"Center of the first cluster (red) after one iteration: {centers\_iteration\_1[0]}")

print(f"Center of the second cluster (green) after two iterations: {centers\_iteration\_2[1]}")

print(f"Center of the third cluster (blue) when clustering converges: {converged\_centers[2]}")

print(f"Number of iterations required for convergence: {num\_iterations\_to\_converge}")

**3.** **Build a Binary**[**Decision Trees**](http://lms.nmit.ac.in/moodle/mod/lesson/view.php?id=24467)**using zoo data available at UCI Zoo Data Set. Generate a confusion matrix and print class wise accuracy, precision and recall in your result.**

**Given csv file: zoo.csv**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import confusion\_matrix, classification\_report

data = pd.read\_csv('zoo\_data.csv')

# Split the dataset into features and target variable

X = data.iloc[:, :-1]

y = data.iloc[:, -1]

# Split the dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)

# Create a Decision Tree Classifier object

clf = DecisionTreeClassifier()

# Train the model using the training sets

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

# Predict the response for test dataset

y\_pred = clf.predict(X\_test)

# Create a confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Print the confusion matrix

print("Confusion Matrix:")

print(cm)

# Print the classification report

cr = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(cr)

# Visualize the Decision Tree with a simplified text representation

from sklearn.tree import export\_text

tree\_text = export\_text(clf, feature\_names=list(X.columns))

print("Decision Tree:")

print(tree\_text)

**4. Use an appropriate 2-dimensional data set and generate scatter plots of its features. Build a correlation matrix and use linear regression to compute the regression parameters. Also compute the Cost, SSE, SSR, SST and R2.**

**Given csv file.**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.linear\_model import LinearRegression

# Load data from the CSV file without column names

data = pd.read\_csv('Food-Truck-LineReg.csv')

# Assign generic column names

data.columns = ['X', 'Y']

# Extract the X and Y variables from the dataset

X = data[['X']].values

Y = data[['Y']].values

# Create and fit the LinearRegression model

model = LinearRegression().fit(X, Y)

# Scatter plot

plt.scatter(X, Y)

plt.plot(X, model.predict(X), color='red') # Plot the regression line

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Scatter Plot with Regression Line")

plt.show()

# Get regression parameters

intercept, slope = model.intercept\_[0], model.coef\_[0][0]

# Calculate Cost, SSE, SSR, SST, and R-squared

Y\_pred = model.predict(X)

Cost = np.mean((Y\_pred - Y) \*\* 2)

SSE = np.sum((Y - Y\_pred) \*\* 2)

SSR = np.sum((Y\_pred - np.mean(Y)) \*\* 2)

SST = np.sum((Y - np.mean(Y)) \*\* 2)

R\_squared = SSR / SST

correlation\_matrix = data.corr()

print("Regression Parameters:")

print(f"Intercept (b0): {intercept}")

print(f"Slope (b1): {slope}")

print("Cost:", Cost)

print("SSE:", SSE)

print("SSR:", SSR)

print("SST:", SST)

print("R-squared:", R\_squared)

print("Correlation Matrix: ",correlation\_matrix)

**5. Use an appropriate multi-dimensional data set to perform Logistic regression for multi-class classification. Illustrate the gradient descent method and compute the regression parameters. Also demonstrate the effect of feature pre-processing like removal of noise, NANs and missing value imputation.**

**Given csv file.**

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

import numpy as np

import matplotlib.pyplot as plt

# Load the local dataset

df = pd.read\_csv('Student-University.csv')

#Adding columns

df.columns=['X','y','Target']

# Assuming 'X' and 'y' are the feature and target columns

X = df[['X', 'y']].values

y = df['Target'].values

# Split the dataset into training and testing sets

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

# Standardize the features

scaler = StandardScaler()

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

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

# Initialize the logistic regression model

model = LogisticRegression(multi\_class='multinomial', solver='lbfgs')

# Train the model on the training data

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

# Predict classes on the test set

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

# Calculate accuracy

accuracy = (y\_pred == y\_test).mean()

print(f"Accuracy: {accuracy}")

# Gradient Descent for Logistic Regression

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Generate x values

x = np.linspace(-7, 7, 200)

# Calculate corresponding y values using the sigmoid function

y = sigmoid(x)

# Plot the sigmoid function

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

plt.plot(x, y, label='Sigmoid Function', color='b')

plt.xlabel('x')

plt.ylabel('sigmoid(x)')

plt.title('Sigmoid Function')

plt.grid(True)

plt.legend()

plt.show()

# Initialize parameters

theta = np.zeros(X\_train.shape[1]) # Initialize parameters to zeros

alpha = 0.01 # Learning rate

num\_iterations = 1000

# Perform gradient descent

for \_ in range(num\_iterations):

z = np.dot(X\_train, theta)

h = sigmoid(z)

error = h - y\_train

gradient = np.dot(X\_train.T, error) / len(y\_train)

theta -= alpha \* gradient

# Print the computed regression parameters

print(f'Regression Parameters using Gradient Descent: {theta}')

print("Original Data:")

print(df)

# Step 1: Removing Noise

# In this example, we'll consider any value above 10 as noise and remove it.

df = df[df['y'] <= 10]

# Step 2: Handling Missing Values (NaNs)

# In this example, we'll use mean imputation to fill missing values.

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

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

# Display the cleaned dataset

print("\nCleaned Data:")

print(df)