Week 5 & 6 Programs

22. Write a Python program to read the data randomly of 100 samples and split the data into 20% testing data and display it without using sklearn.

```
'''22. Python program to read the data randomly of 100 samples and split the data into 20% testing data and display it without using sklearn '''

import random

# Function to read data and split it

def split_data(data, test_ratio=0.2):

    # Shuffle the data randomly

    random.shuffle(data)

    # Calculate the index for the test split

    test_size = int(len(data) * test_ratio)

    # Split the data into training and testing

    test_data = data[:test_size]

    train_data = data[test_size:]

    return train_data, test_data

# Example of reading 100 sample data (for demonstration)

# Let's assume you have a dataset of 100 samples.

data = [i for i in range(1, 101)]                    # A simple dataset of numbers 1 to 100

# Call the function to split the data

train_data, test_data = split_data(data)

# Display the test data

print("Testing Data (20%):")

print(test_data)
```

23. Write a Python program to create dataframe using random data and draw a plot (Clasification) without using sklearn.

```
'''23. Python program to create dataframe using random data and draw a plot
(Clasification) without using sklearn '''

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Function to create random dataset for classification

def create_random_data(num_samples=100):

    X1 = np.random.randn(num_samples)          # Random values

    X2 = np.random.randn(num_samples)

    # Create a label (Y) based on a simple condition for classification

    # For simplicity, let's assume class 0 if X1 + X2 < 0, else class 1

    Y = np.where(X1 + X2 < 0, 0, 1)

    # Create a DataFrame from the data

    df = pd.DataFrame({

        'X1': X1,

        'X2': X2,

        'Label': Y

    })

    return df

# Function to plot the classification data

def plot_classification(df):

    # Scatter plot for the classification data

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

    # Plot points where Label == 0 (Class 0)
```

```
    class_0 = df[df['Label'] == 0]
    plt.scatter(class_0['X1'], class_0['X2'], color='blue', label='Class 0',alpha=0.6)
    # Plot points where Label == 1 (Class 1)
    class_1 = df[df['Label'] == 1]
    plt.scatter(class_1['X1'], class_1['X2'], color='red', label='Class 1', alpha=0.6)
    # Set the plot title and labels
    plt.title('Random Classification Data')
    plt.xlabel('Feature X1')
    plt.ylabel('Feature X2')
    plt.legend()
    # Show the plot
    plt.show()
# Main function
def main():
    # Create random classification data
    df = create_random_data(100)
    # Display the first few rows of the dataframe
    print(df.head())
    # Plot the data
    plot_classification(df)
if __name__ == "__main__":
    main()
```

24. Write a Python program to create dataframe using random data and draw a plot for test data (Clasification) without using sklearn.

```
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
# Function to create random dataset for classification
def create_random_data(num_samples=100):
    X1 = np.random.randn(num_samples)  # Random values
    X2 = np.random.randn(num_samples)
    # Create a label (Y) based on a simple condition for classification
    # For simplicity, let's assume class 0 if X1 + X2 < 0, else class 1
    Y = np.where(X1 + X2 < 0, 0, 1)
    # Create a DataFrame from the data
    df = pd.DataFrame({
        'X1': X1,
        'X2': X2,
        'Label': Y
    })
    return df
# Function to read data and split it
def split_data(data, test_ratio=0.2):
    # Calculate the index for the test split
    test_size = int(len(data) * test_ratio)
    # Split the data into training and testing
    test_data = data[:test_size]
    train_data = data[test_size:]
    return train_data, test_data
```

```
# Function to plot the classification data
def plot_classification(df):
    # Scatter plot for the classification data
    plt.figure(figsize=(8, 6))
    # Plot points where Label == 0 (Class 0)
    class_0 = df[df['Label'] == 0]
    plt.scatter(class_0['X1'], class_0['X2'], color='blue', label='Class 0', alpha=0.6)
    # Plot points where Label == 1 (Class 1)
    class_1 = df[df['Label'] == 1]
    plt.scatter(class_1['X1'], class_1['X2'], color='red', label='Class 1', alpha=0.6)
    # Set the plot title and labels
    plt.title('Random Classification Data')
    plt.xlabel('Feature X1')
    plt.ylabel('Feature X2')
    plt.legend()
    # Show the plot
    plt.show()
# Main function
def main():
    # Create random classification data
    df = create_random_data(100)
    train_data,test_data = split_data(df)
    # Plot the data
    plot_classification(test_data)
if __name__ == "__main__":
    main()
```

26. Split a dataset into 80% training and 20% test sets using Scikit learn's train_test_split().

```
'''26. Split a diabetes dataset into 80% training and 20% test sets using Scikit
learn's train_test_split(). '''
'''This dataset consists of 442 instances, each with 10 features that
represent various medical measurements related to diabetes progression.'''
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
import pandas as pd
# Load Diabetes dataset
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target
# Split the dataset into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print(len(X_train))
print(len(X_test))
print(len(y_train))
print(len(y_test))
# Create DataFrame for visualization (estoptional)
df = pd.DataFrame(X_train, columns=diabetes.feature_names)
df['target'] = y_train
print(df.head())
```

Week 5 Programs

27. Experiment with different test_size values (e.g., 0.2, 0.3, 0.4) and observe model performance.

```
'''27. Experiment with different test_size values (e.g., 0.2, 0.3, 0.4) and observe model
performance on diabetes dataset'''

import numpy as np

from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

# Load Diabetes dataset

diabetes = load_diabetes()

X = diabetes.data

y = diabetes.target

# Function to train the model and evaluate performance

def evaluate_model(test_size):

    # Split the dataset into training and testing sets

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=42)

    # Initialize the model

    model = LinearRegression()

    # Train the model

    model.fit(X_train, y_train)

    # Make predictions

    y_pred = model.predict(X_test)

    # Calculate the Mean Squared Error (MSE) for model performance

    mse = mean_squared_error(y_test, y_pred)

    return mse
```

```
# Experiment with different test_size values: 0.2, 0.3, 0.4
test_sizes = [0.2, 0.3, 0.4]
results = {}
for size in test_sizes:
    mse = evaluate_model(size)
    results[size] = mse
    print(f"Test size: {size}, Mean Squared Error: {mse:.4f}")
# Comparison of model performance
print("\nComparison of Model Performance:")
for size, mse in results.items():
    print(f"Test size: {size} -> MSE: {mse:.4f}")
```

28. Use a fixed random_state value for reproducibility when splitting data, and test with different random_state values.

'''28. Use a fixed random_state value for reproducibility when splitting data, and test with different random_state values. '''

import numpy as np

from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

Load Diabetes dataset

diabetes = load_diabetes()

X = diabetes.data

y = diabetes.target

Function to train the model and evaluate performance

def evaluate_model(random_state_value):

    # Split the dataset into training and testing sets with a specific random_state

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=random_state_value)

    # Initialize the model

    model = LinearRegression()

    # Train the model

    model.fit(X_train, y_train)

    # Make predictions

    y_pred = model.predict(X_test)

    # Calculate the Mean Squared Error (MSE) for model performance

    mse = mean_squared_error(y_test, y_pred)

    return mse

Experiment with different random_state values: 42, 0, 100, None

random_state_values = [42, 0, 100, None]

results = {}

```python
for state in random_state_values:
    mse = evaluate_model(state)
    results[state] = mse
    print(f"Random State: {state}, Mean Squared Error: {mse:.4f}")

# Comparison of model performance for different random states
print("\nComparison of Model Performance:")
for state, mse in results.items():
    print(f"Random State: {state} -> MSE: {mse:.4f}")
```

29. Clean and preprocess a dataset (handle missing values, scale features) before splitting into training and test sets.

'''29. Clean and preprocess a dataset (handle missing values, scale

features) before splitting into training and test sets'''

import numpy as np

import pandas as pd

from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.impute import SimpleImputer

 # Load the Diabetes dataset

diabetes = load_diabetes()

X = pd.DataFrame(diabetes.data, columns=diabetes.feature_names)

y = pd.Series(diabetes.target)

 # Step 1: Check for missing values

print("Missing values in each feature:")

print(X.isnull().sum())

 # Step 2: Handle missing values by imputation (if any)

For demonstration, let's assume some missing values are introduced randomly

Introduce missing values randomly in 5% of the data for each feature

np.random.seed(42)

missing_rate = 0.05

n_missing = int(missing_rate * X.size)

missing_indices = np.random.choice(X.size, n_missing, replace=False)

X.values.ravel()[missing_indices] = np.nan

Impute missing values using the mean strategy

imputer = SimpleImputer(strategy='mean')

X_imputed = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)

 # Step 3: Scale the features using Standardization

scaler = StandardScaler()

X_scaled = pd.DataFrame(scaler.fit_transform(X_imputed), columns=X.columns)

 # Step 4: Split into training and test sets (80% training, 20% test)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

 # Show the preprocessed data

print("\nPreprocessed Training Features (first 5 rows):")

print(X_train.head())

print("\nPreprocessed Test Features (first 5 rows):")

print(X_test.head())

30. Compare results when applying feature scaling before or after splitting the data.

'''30. Compare results when applying feature scaling before or after splitting the data. '''

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import LogisticRegression

from sklearn.datasets import load_diabetes

from sklearn.metrics import accuracy_score

 # Load the diabetes dataset

data = load_diabetes()

X = data.data

y = (data.target > np.median(data.target)).astype(int) # Convert target to binary as 0 or 1

 # Function to train and evaluate model with scaling before splitting

def model_with_scaling_before_split(X, y):

```
    # Split the data 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)
    # Apply feature scaling to the entire dataset before splitting
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)  # Scale the whole dataset
    # Split the scaled data
    X_train_scaled, X_test_scaled = X_scaled[:len(X_train)], X_scaled[len(X_train):]
    # Train the model
    model = LogisticRegression(max_iter=200)
    model.fit(X_train_scaled, y_train)
    # Predict and evaluate the model
    y_pred = model.predict(X_test_scaled)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy
# Function to train and evaluate model with scaling after splitting
def model_with_scaling_after_split(X, y):
    # Split the data into training and testing sets (80% train, 20% test)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    # Apply feature scaling after splitting
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)  # Scale only the training set
    X_test_scaled = scaler.transform(X_test)  # Transform the test set based on training data
    # Train the model
    model = LogisticRegression(max_iter=200)
    model.fit(X_train_scaled, y_train)
    # Predict and evaluate the model
    y_pred = model.predict(X_test_scaled)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy
```

 # Compare the results

```
accuracy_before_split = model_with_scaling_before_split(X, y)
accuracy_after_split = model_with_scaling_after_split(X, y)
print(f"Accuracy with scaling before splitting: {accuracy_before_split:.4f}")
print(f"Accuracy with scaling after splitting: {accuracy_after_split:.4f}")
```

Assessment 3:

Write a Python program using Scikit-learn to split the iris dataset into 70% train data and 30% test data. Out of total 150 records, the training set will contain 120 records and the test set contains 30 of those records. Print both datasets.

(Students only must write the program for this Assessment Question and should execute in Lab)