Sri Sivasubramaniya Nadar College of Engineering, Chennai
(An autonomous Institution affiliated to Anna University)

Degree & Branch	B.E. Computer Science & Engineering	Semester	V
Subject Code & Name	ICS1512 & Machine Learning Algorithms Laboratory		
Academic year	2025-2026 (Odd)	Batch:2023-2028	**Due date:**

Experiment 1 : Working with Python packages– Numpy, Scipy, Scikit-learn, Mat plotlib

1. Aim

To understand and explore essential Python libraries used in machine learning, such as `pandas`, `numpy`, `matplotlib`, `seaborn`, and `scikit-learn`. This includes learning how to handle, visualize, and preprocess datasets effectively for model building.

2. Libraries Used

- **NumPy:** A fundamental package for numerical computing in Python. It provides support for arrays, matrices, and mathematical functions.
  - Common methods: `np.array()`, `np.arange()`, `np.mean()`, `np.std()`, `np.dot()`
- **Pandas:** A powerful data manipulation and analysis library. It offers data structures like Series and DataFrame.
  - Common methods: `pd.read_csv()`, `df.head()`, `df.describe()`, `df.groupby()`, `df.isnull()`, `df.fillna()`
- **Matplotlib:** A plotting library used for creating static, animated, and interactive visualizations in Python.
  - Common methods: `plt.plot()`, `plt.hist()`, `plt.title()`, `plt.xlabel()`, `plt.ylabel()`
- **Seaborn:** A statistical data visualization library based on Matplotlib. It provides high-level functions for attractive and informative graphics.
  - Common methods: `sns.heatmap()`, `sns.pairplot()`, `sns.boxplot()`, `sns.countplot()`
- **Scikit-learn:** A machine learning library that provides simple and efficient tools for data mining and data analysis.
  - Common methods: `train_test_split()`, `fit()`, `predict()`, `accuracy_score()`, `StandardScaler()`, `LogisticRegression()`
- **SciPy:** A library used for scientific and technical computing. It builds on NumPy and provides modules for optimization, integration, interpolation, linear algebra, and statistics.

  - Common methods: `scipy.stats.norm()`, `scipy.optimize.curve_fit()`, `scipy.integrate.quad()`, `scipy.linalg.inv()`, `scipy.signal.find_peaks()`

3. Mathematical/Theoretical Description of the Algorithm/Objective Performed

3.1 Handling Missing Values

Missing values can negatively impact the performance of machine learning models by:

- Distorting statistical summaries
- Causing errors in training algorithms
- Leading to biased predictions

So, it is crucial to detect and properly handle them before modeling. There are several ways to handle missing values:

- Missing values can be handled using imputation techniques such as replacing them with the **mean**, **median**, or **mode** using the `fillna()` method in pandas.
- If a column has a large number of missing values and does not contribute significantly to the prediction task, it can be dropped. This reduces noise and improves model efficiency.

3.2 Label Encoding

To train machine learning models, input features must be in numeric format. Categorical variables (e.g., "Yes"/"No", "Graduate"/"Not Graduate") need to be converted into numbers.

- Binary categorical values can be mapped directly, e.g., "Yes" to 1 and "No" to 0, ensuring compatibility with ML models.
- For features with more than two categories, **one-hot encoding** is used to avoid introducing ordinal relationships. Each category becomes a separate binary column.

3.3 Plotting

Data visualization helps identify patterns, distributions, and outliers.

- `heatmap()`: Visualizes correlation between numerical features using color intensity. Useful for detecting interdependencies.
- `hist()`: Displays frequency distribution of a numeric variable. Helps detect skewness and missing value gaps.
- `boxplot()`: Highlights spread using quartiles and identifies outliers. Good for checking symmetry and extreme values.

3.4 Removal of Outliers

Outliers, once detected using boxplots, can be handled in the following ways:

- **Deletion:** Drop rows with outlier values beyond a threshold (commonly $1.5 \times \text{IQR}$). Reduces noise but may result in data loss.
- **Imputation:** Replace outliers with column **mean** or **median**. Preserves dataset size and balances distributions. Median is preferred for skewed data.

3.5 Standardization

Standardization scales features to have a mean of 0 and standard deviation of 1. It ensures all features contribute equally, improving model performance.

- Particularly important for algorithms like *Logistic Regression* and *KNN*.
- The formula for standardization is:
$$z = \frac{x - \mu}{\sigma}$$
  where x is the original value, μ is the mean, and σ is the standard deviation.

The preprocessing steps included handling missing values, encoding categorical variables, visualizing data using heatmaps, histograms, and boxplots, addressing outliers, and applying standardization to bring all features to a common scale—thereby ensuring stable and efficient model training.

4. Code

Numpy

Code Cell

```
import numpy as np
```

Code Cell

```
# 1. Create a 1-dimensional NumPy array (vector)
vector_array = np.array([1, 2, 3, 4, 5])
print("1-dimensional array (vector):")
print(vector_array)
print("Shape:", vector_array.shape)
print("-" * 20)

# 2. Create a 2-dimensional NumPy array (matrix)
matrix_array = np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])
print("2-dimensional array (matrix):")
print(matrix_array)
print("Shape:", matrix_array.shape)
print("-" * 20)
```

```
# 3. Create a NumPy array filled with zeros
zeros_array = np.zeros((2, 3))
print("Array filled with zeros (2x3):")
print(zeros_array)
print("Shape:", zeros_array.shape)
print("-" * 20)

# 4. Create a NumPy array filled with ones
ones_array = np.ones((3, 2))
print("Array filled with ones (3x2):")
print(ones_array)
print("Shape:", ones_array.shape)
print("-" * 20)

# 5. Create a NumPy array with a range of values
range_array = np.arange(0, 10, 2)
print("Array with a range of values (0 to 10, step 2):")
print(range_array)
print("Shape:", range_array.shape)
print("-" * 20)

# 6. Create a NumPy array with evenly spaced values
linspace_array = np.linspace(0, 1, 5)
print("Array with evenly spaced values (0 to 1, 5 samples):")
print(linspace_array)
print("Shape:", linspace_array.shape)
print("-" * 20)
```

Output

```
1-dimensional array (vector):
[1 2 3 4 5]
Shape: (5,)
--------------------
2-dimensional array (matrix):
[[1.1 2.2 3.3]
 [4.4 5.5 6.6]]
Shape: (2, 3)
--------------------
Array filled with zeros (2x3):
[[0. 0. 0.]
 [0. 0. 0.]]
Shape: (2, 3)
--------------------
Array filled with ones (3x2):
[[1. 1.]
 [1. 1.]
```

```
 [1. 1.]]
Shape: (3, 2)
--------------------
Array with a range of values (0 to 10, step 2):
[0 2 4 6 8]
Shape: (5,)
--------------------
Array with evenly spaced values (0 to 1, 5 samples):
[0.   0.25 0.5  0.75 1.  ]
Shape: (5,)
--------------------
```

Basic Arithmetic operations

Code Cell

```
# Create two NumPy arrays of the same shape (2x3)
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8, 9], [10, 11, 12]])

print("Array 1:")
print(arr1)
print("\nArray 2:")
print(arr2)

# Perform element-wise addition
addition_result = arr1 + arr2
print("\nElement-wise Addition:")
print(addition_result)

# Perform element-wise subtraction
subtraction_result = arr1 - arr2
print("\nElement-wise Subtraction:")
print(subtraction_result)

# Perform element-wise multiplication
multiplication_result = arr1 * arr2
print("\nElement-wise Multiplication:")
print(multiplication_result)

# Perform element-wise division
division_result = arr1 / arr2
print("\nElement-wise Division:")
print(division_result)
```

Output

```
Array 1:
```

```
[[1 2 3]
 [4 5 6]]

Array 2:
[[ 7  8  9]
 [10 11 12]]

Element-wise Addition:
[[ 8 10 12]
 [14 16 18]]

Element-wise Subtraction:
[[-6 -6 -6]
 [-6 -6 -6]]

Element-wise Multiplication:
[[ 7 16 27]
 [40 55 72]]

Element-wise Division:
[[0.14285714 0.25       0.33333333]
 [0.4        0.45454545 0.5       ]]
```

Array Manipulations

Code Cell

```
# 1. Create a NumPy array, a 2x3 matrix
original_array = np.array([[1, 2, 3], [4, 5, 6]])
print("Original array (2x3):")
print(original_array)
print("-" * 20)

# 2. Reshape the created array into a different shape, a 3x2 matrix
reshaped_array = original_array.reshape(3, 2)
print("Reshaped array (3x2):")
print(reshaped_array)
print("-" * 20)

# 3. Transpose the original array
transposed_array = original_array.T
print("Transposed original array (3x2):")
print(transposed_array)
```

Output

```
Original array (2x3):
[[1 2 3]
```

```
 [4 5 6]]
--------------------
Reshaped array (3x2):
[[1 2]
 [3 4]
 [5 6]]
--------------------
Transposed original array (3x2):
[[1 4]
 [2 5]
 [3 6]]
```

Array Indexing and Slicing

Code Cell

```
# 1. Create a 1-dimensional NumPy array (vector) with at least 5 elements.
vector_array = np.array([10, 20, 30, 40, 50, 60])
print("1D Array:")
print(vector_array)
print("-" * 20)

# 2. Access and print a single element from the 1D array using positive indexing.
print("Element at index 2 (positive indexing):", vector_array[2])
print("-" * 20)

# 3. Access and print a single element from the 1D array using negative indexing.
print("Element at index -1 (negative indexing):", vector_array[-1])
print("-" * 20)

# 4. Slice and print a portion of the 1D array.
print("Slice from index 1 to 4 (exclusive of 4):", vector_array[1:4])
print("-" * 20)

# 5. Create a 2-dimensional NumPy array (matrix) with at least 3 rows and 3 columns.
matrix_array = np.array([[1, 2, 3],
                         [4, 5, 6],
                         [7, 8, 9],
                         [10, 11, 12]])
print("2D Array (Matrix):")
print(matrix_array)
print("-" * 20)

# 6. Access and print a single element from the 2D array using row and column indices.
print("Element at row 1, column 2:", matrix_array[1, 2])
print("-" * 20)

# 7. Slice and print a portion of the 2D array (e.g., a sub-matrix).
```

```
print("Sub-matrix from row 1 to 3 (exclusive of 3), column 0 to 2 (exclusive of 2):")
print(matrix_array[1:3, 0:2])
print("-" * 20)

# 8. Slice and print a specific row from the 2D array.
print("Third row:")
print(matrix_array[2, :])
print("-" * 20)

# 9. Slice and print a specific column from the 2D array.
print("Second column:")
print(matrix_array[:, 1])
```

Output

```
1D Array:
[10 20 30 40 50 60]
--------------------
Element at index 2 (positive indexing): 30
--------------------
Element at index -1 (negative indexing): 60
--------------------
Slice from index 1 to 4 (exclusive of 4): [20 30 40]
--------------------
2D Array (Matrix):
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
--------------------
Element at row 1, column 2: 6
--------------------
Sub-matrix from row 1 to 3 (exclusive of 3), column 0 to 2 (exclusive of 2):
[[4 5]
 [7 8]]
--------------------
Third row:
[7 8 9]
--------------------
Second column:
[ 2  5  8 11]
```

Array concatenation

Code Cell

```
# Create two NumPy arrays
arr1 = np.array([[1, 2], [3, 4]])
```

```
arr2 = np.array([[5, 6], [7, 8]])

print("Array 1:")
print(arr1)
print("\nArray 2:")
print(arr2)

# Concatenate vertically (along axis 0)
vertical_concatenation = np.concatenate((arr1, arr2), axis=0)
print("\nVertical Concatenation:")
print(vertical_concatenation)

# Concatenate horizontally (along axis 1)
horizontal_concatenation = np.concatenate((arr1, arr2), axis=1)
print("\nHorizontal Concatenation:")
print(horizontal_concatenation)
```

Output

```
Array 1:
[[1 2]
 [3 4]]

Array 2:
[[5 6]
 [7 8]]

Vertical Concatenation:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

Horizontal Concatenation:
[[1 2 5 6]
 [3 4 7 8]]
```

Pandas

Code Cell

```
import pandas as pd
```

Series

Code Cell

```
list_data = [10, 20, 30, 40, 50]
```

```
series_from_list = pd.Series(list_data)
dict_data = {'a': 100, 'b': 200, 'c': 300, 'd': 400}
series_from_dict = pd.Series(dict_data)

print("Series from list:")
print(series_from_list)
print("\nSeries from dictionary:")
print(series_from_dict)
```

Output

```
Series from list:
0    10
1    20
2    30
3    40
4    50
dtype: int64

Series from dictionary:
a    100
b    200
c    300
d    400
dtype: int64
```

Dataframe

Code Cell

```
data = {
    'Column A': [1, 2, 3, 4, 5],
    'Column B': ['A', 'B', 'C', 'D', 'E'],
    'Column C': [10.1, 20.2, 30.3, 40.4, 50.5]
}

df = pd.DataFrame(data)
display(df)
```

Output

```
   Column A Column B  Column C
0         1        A      10.1
1         2        B      20.2
2         3        C      30.3
3         4        D      40.4
4         5        E      50.5
```

Basic Operations

Code Cell

```
# 1. Select and print a single column from the DataFrame df using its column label.
print("Selecting 'Column A':")
display(df['Column A'])
print("-" * 20)

# 2. Select and print multiple columns from the DataFrame df using a list of column labels.
print("Selecting 'Column A' and 'Column C':")
display(df[['Column A', 'Column C']])
print("-" * 20)

# 3. Filter the DataFrame df to select and print rows where the value in 'Column A' is greater
print("Filtering rows where 'Column A' > 2:")
display(df[df['Column A'] > 2])
print("-" * 20)

# 4. Filter the DataFrame df to select and print rows based on a condition on a non-numeric col
print("Filtering rows where 'Column B' is 'C':")
display(df[df['Column B'] == 'C'])
print("-" * 20)

# 5. Add a new column named 'Column D' to the DataFrame df with values that are the sum of 'Col
df['Column D'] = df['Column A'] + df['Column C']

# 6. Add another new column named 'Column E' to the DataFrame df with some example string or bo
df['Column E'] = ['True', 'False', 'True', 'False', 'True']

# 7. Print the DataFrame df to show the newly added columns.
print("DataFrame with new columns 'Column D' and 'Column E':")
display(df)
```

Output

```
Selecting 'Column A':
0    1
1    2
2    3
3    4
4    5
Name: Column A, dtype: int64--------------------
Selecting 'Column A' and 'Column C':
   Column A  Column C
0         1      10.1
1         2      20.2
2         3      30.3
```

```
3        4      40.4
4        5      50.5--------------------
Filtering rows where 'Column A' > 2:
   Column A Column B  Column C
2        3        C      30.3
3        4        D      40.4
4        5        E      50.5--------------------
Filtering rows where 'Column B' is 'C':
   Column A Column B  Column C
2        3        C      30.3--------------------
DataFrame with new columns 'Column D' and 'Column E':
   Column A Column B  Column C  Column D Column E
0        1        A      10.1      11.1     True
1        2        B      20.2      22.2    False
2        3        C      30.3      33.3     True
3        4        D      40.4      44.4    False
4        5        E      50.5      55.5     True
```

Handling Missing values

Code Cell

```
# 1. Introduce missing values into the DataFrame
df.loc[1, 'Column A'] = np.nan
df.loc[3, 'Column C'] = np.nan
df.loc[2, 'Column E'] = np.nan
print("DataFrame with missing values:")
display(df)
print("-" * 20)

# 2. Identify and print the locations of missing values
print("Locations of missing values (True means missing):")
display(df.isnull())
print("-" * 20)

# 3. Count and print the number of missing values per column
print("Number of missing values per column:")
print(df.isnull().sum())
print("-" * 20)

# 4. Drop rows with missing values
print("DataFrame after dropping rows with missing values:")
display(df.dropna())
print("-" * 20)

# 5. Fill missing values with a specific value (e.g., 0)
print("DataFrame after filling missing values with 0:")
display(df.fillna(0))
```

```
print("-" * 20)

# Example of filling with the mean (for a numeric column)
df_filled_mean = df.copy()
mean_value = df_filled_mean['Column C'].mean()
df_filled_mean['Column C'].fillna(mean_value, inplace=True)
print("DataFrame after filling 'Column C' missing values with the column mean:")
display(df_filled_mean)
```

Output

```
DataFrame with missing values:
   Column A Column B  Column C  Column D Column E
0       1.0        A      10.1      11.1     True
1       NaN        B      20.2      22.2    False
2       3.0        C      30.3      33.3      NaN
3       4.0        D       NaN      44.4    False
4       5.0        E      50.5      55.5     True--------------------
Locations of missing values (True means missing):
   Column A  Column B  Column C  Column D  Column E
0     False     False     False     False     False
1      True     False     False     False     False
2     False     False     False     False      True
3     False     False      True     False     False
4     False     False     False     False     False--------------------
Number of missing values per column:
Column A    1
Column B    0
Column C    1
Column D    0
Column E    1
dtype: int64
--------------------
DataFrame after dropping rows with missing values:
   Column A Column B  Column C  Column D Column E
0       1.0        A      10.1      11.1     True
4       5.0        E      50.5      55.5     True--------------------
DataFrame after filling missing values with 0:
   Column A Column B  Column C  Column D Column E
0       1.0        A      10.1      11.1     True
1       0.0        B      20.2      22.2    False
2       3.0        C      30.3      33.3        0
3       4.0        D       0.0      44.4    False
4       5.0        E      50.5      55.5     True--------------------
DataFrame after filling 'Column C' missing values with the column mean:
/tmp/ipython-input-14-4219056867.py:32: FutureWarning: A value is trying to be set on a copy o
The behavior will change in pandas 3.0. This inplace method will never work because the interm
```

```
For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: valu


  df_filled_mean['Column C'].fillna(mean_value, inplace=True)
   Column A Column B  Column C  Column D Column E
0       1.0        A    10.100      11.1     True
1       NaN        B    20.200      22.2    False
2       3.0        C    30.300      33.3      NaN
3       4.0        D    27.775      44.4    False
4       5.0        E    50.500      55.5     True
```

Group by and Describe methods

Code Cell

```
# 1. Calculate and print the descriptive statistics for the numeric columns
print("Descriptive Statistics:")
display(df.describe())
print("-" * 20)

# 2. Group by 'Column B' and calculate the mean of numeric columns
print("Mean of numeric columns grouped by 'Column B':")
display(df.groupby('Column B').mean(numeric_only=True))
```

Output

```
Descriptive Statistics:
       Column A   Column C   Column D
count  4.000000   4.000000   5.000000
mean   3.250000  27.775000  33.300000
std    1.707825  17.249034  17.550641
min    1.000000  10.100000  11.100000
25%    2.500000  17.675000  22.200000
50%    3.500000  25.250000  33.300000
75%    4.250000  35.350000  44.400000
max    5.000000  50.500000  55.500000--------------------
Mean of numeric columns grouped by 'Column B':
          Column A  Column C  Column D
Column B
A              1.0      10.1      11.1
B              NaN      20.2      22.2
C              3.0      30.3      33.3
D              4.0       NaN      44.4
E              5.0      50.5      55.5
```

Scipy

Code Cell

```
from scipy import stats
import numpy as np

data = np.array([15, 22, 18, 25, 30, 12, 20, 28, 19, 24])

print("Sample Data:", data)
print("-" * 20)

print("Mean:", np.mean(data))
print("Median:", np.median(data))
print("Mode:", stats.mode(data).mode)
print("Standard Deviation:", np.std(data))
print("Variance:", np.var(data))
print("Skewness:", stats.skew(data))
print("Kurtosis:", stats.kurtosis(data))
print("25th Percentile:", np.percentile(data, 25))
print("75th Percentile:", np.percentile(data, 75))
print("Interquartile Range (IQR):", stats.iqr(data))
```

Output

```
Sample Data: [15 22 18 25 30 12 20 28 19 24]
--------------------
Mean: 21.3
Median: 21.0
Mode: 12
Standard Deviation: 5.348831648126533
Variance: 28.610000000000003
Skewness: -0.048853142596844173
Kurtosis: -0.8998823625854135
25th Percentile: 18.25
75th Percentile: 24.75
Interquartile Range (IQR): 6.5
```

5. Results and Discussions

Digit Recognition using MNIST Dataset

This problem focuses on recognizing handwritten digits (0–9) from grayscale images. Convolutional Neural Networks (CNNs) are highly suitable due to their ability to extract spatial features from images. Alternatively, Support Vector Machines (SVMs) can be applied after flattening image data and applying feature scaling.

Loan Amount Prediction

This is a regression task where the objective is to estimate the loan amount based on various applicant and property attributes. Linear Regression is commonly used here due to the continuous target variable. Prior to modeling, steps like outlier treatment, missing value imputation, and feature normalization are crucial for improving accuracy.

Iris Species Classification

The Iris dataset is a well-known example of a multi-class classification task, where the aim is to predict the species of an iris flower based on petal and sepal measurements. Algorithms like K-Nearest Neighbors (KNN) and Support Vector Machines (SVM) perform well on this dataset, especially when features are standardized.

Diabetes Detection

This is a binary classification problem where the model predicts whether a patient is likely to have diabetes based on health-related metrics. Logistic Regression and SVMs are appropriate choices, and preprocessing techniques like normalization and outlier detection help improve model reliability.

Spam Email Detection

In this task, the objective is to classify emails as spam or not based on textual features. Logistic Regression and SVM are effective due to their performance in high-dimensional spaces. Feature extraction using methods like TF-IDF (Term Frequency-Inverse Document Frequency) transforms raw text into a usable numerical format for training models.

Dataset	**Type of ML Task**	**Suitable ML Algorithm**
Iris Dataset	Multi-class Classification	KNN, SVM
Loan Amount Prediction	Regression	Linear Regression
Predicting Diabetes	Binary Classification	SVM, XGBoost
Email Spam Classification	Binary Classification	Logistic Regression, SVM
Handwritten Digit Recognition	Multi-class Classification	CNN, SVM

Table 1: ML Task and Suitable Algorithms for Different Datasets

6. Learning Practices

- **Dealing with Missing Data:** Incomplete values were handled using imputation techniques such as mean or median substitution, or by dropping non-essential features to retain data quality.
- **Converting Categorical Data:** Non-numeric features were transformed into numerical form using label encoding and one-hot encoding, ensuring that models could interpret them effectively.
- **Evaluating Feature Importance:** Tools like correlation matrices, heatmaps, and boxplots were used to identify key features, detect redundant variables, and highlight outliers.

- **Choosing the Right Model:** Depending on whether the task was classification or regression, models were chosen based on input type, number of features, and the nature of the output.