**5KS08 EMERGING TECHNOLOGY LAB – I**

**Data Science and Statistics Lab - I**

**Course Prerequisite:** Basic knowledge of Mathematics.

**Course Objectives:**

Throughout the course, students will be expected to demonstrate their understanding of Data

Science and Statistics by being able to do each of the following:

* Demonstrate knowledge of statistical data analysis techniques utilized in business decision making.
* Apply principles of Data Science to the analysis of business problems.
* Apply the learned concepts for the skillful data management.

**Course Outcomes (Expected Outcome):**

On completion of the course, the students will be able to:

* Demonstrate proficiency with statistical analysis of data.
* Build skills in transformation and merging of data for use in analytic tools.
* Perform linear and multiple linear regression analysis.
* Develop the ability to build and assess data-based models.
* Evaluate outcomes and make decisions based on data.

**List of Experiments based on Syllabus:**

1. Introduction to Basics of Python.
2. Variable creation, Arithmetic and logical operators, Data types and associated operations Sequence data types and associated operations, Strings, Lists, Arrays, Tuples, Dictionary, Sets, Range, Loops and Conditional Statement’s Linear Regression
3. To learn different Libraries in Python and to perform Simple Linear Regression and Multiple Linear Regression.
4. To perform Logistic Regression
5. To perform Linear Discriminate Analysis
6. To perform Quadratic Discriminate Analysis
7. To implement K-Nearest Neighbors technique
8. To learn and perform The Lasso Decision Trees
9. To learn and perform Fitting Classification Trees
10. To learn and perform Fitting Regression Trees.
11. To learn and perform Support Vector Classifier
12. To learn and perform Support Vector Machine

**PRACTICAL-01**

**Aim:**

Introduction to Basics of Python:

**Objective:**

The objective of this experiment is to introduce students to the fundamental concepts of Python programming, including variables, data types, and basic operations. By the end of the experiment, participants should have a basic understanding of Python syntax and be able to write simple Python programs.

**Materials Required:**

1. A computer with Python installed (Python 3.x recommended)

2. Integrated Development Environment (IDE) like IDLE, PyCharm, or Jupyter Notebook

3. Access to relevant Python documentation and resources

**Theory:**

Python is a versatile and widely-used high-level programming language known for its simplicity and readability. It's an interpreted language, allowing developers to write and execute code without the need for a separate compilation step. Python enforces code organization through indentation and whitespace, which is crucial for readability and functionality. In Python, you don't need to explicitly declare variable data types; the language infers them based on the assigned values.

Basic operations like arithmetic are supported and conditional statements like 'if', 'elif' and 'else' enable decision-making in code. Python also offers 'for' and 'while' loops for iteration. Functions, defined using 'def,' allow developers to create reusable code blocks with parameters and return values. Data structures like lists, tuples, and dictionaries provide flexibility for handling collections of data. Python's extensive library ecosystem, including 'numpy' for numerical operations and 'pandas' for data analysis, extends its capabilities. The language supports file handling, exception handling for error management, and object-oriented programming for creating classes and objects. Python's broad applicability spans web development, data science, machine learning, and more, making it a valuable skill for both beginners and experienced programmers.

**Procedure:**

1. Ensure Python is installed on your computer. If not, download and install Python from the official website (https://www.python.org/downloads/).

2. Open your chosen IDE.

3. Start with a brief explanation of Python's print function, which is used to display output on the screen.

4. Discuss the concept of variables in Python and demonstrate how to declare and assign values to variables.

Example:

name = "Alice"

age = 30

height = 1.75

5. Introduce different data types in Python, such as strings, integers, floats, and boolean and explain how to check the data type of a variable using the `type()` function.

Example:

print(type(name)) # Output: <class 'str'>

print(type(age)) # Output: <class 'int'>

print(type(height)) # Output: <class 'float'>

6. Explain and demonstrate basic mathematical operations in Python, including addition, subtraction, multiplication, division, and modulus.

**Example:**

x = 10

y = 3

sum\_result = x + y

difference = x - y

product = x \* y

division\_result = x / y

modulus\_result = x % y

7. Discuss the importance of operator precedence in Python and how to use parentheses to control the order of operations.

Example:

result = (x + y) \* (x - y)

**Conclusion:**

Python is a versatile programming language. It supports basic mathematical operations like addition, subtraction, multiplication, division, and modulus, with control over operator precedence through parentheses.

**PRACTICAL-02**

**Aim:**

Variable creation, Arithmetic and logical operators, Data types and associated operations Sequence data types and associated operations, Strings, Lists, Arrays, Tuples, Dictionary, Sets, Range, Loops and Conditional Statement’s Linear Regression.

**Objective:**

The objective of this experiment is to provide a comprehensive understanding of Python programming, including variable creation, arithmetic and logical operators, data types, sequence data types, strings, lists, arrays, tuples, dictionaries, sets, range, loops, conditional statements, and an introduction to linear regression for data analysis.

**Materials Required:**

1. A computer with Python installed (Python 3.x recommended)

2. Integrated Development Environment (IDE) like IDLE, PyCharm, or Jupyter Notebook

3. Access to relevant Python documentation and resources

**Theory:**

Variable Creation: In Python, you can create variables to store and manipulate data. Variables are dynamically typed, meaning their data type is determined by the value assigned to them.

Arithmetic and Logical Operators: Python supports common arithmetic operations like addition, subtraction, multiplication, and division, as well as logical operators such as AND, OR, and NOT for boolean expressions. These operators are essential for performing calculations and making decisions in your code.

Data Types and Associated Operations: Python provides various data types, including integers, floats, strings, and boolean. Each data type comes with associated operations, such as string concatenation, type conversion, and mathematical operations for numbers.

Sequence Data Types and Associated Operations: Sequences in Python include strings, lists, arrays, and tuples. These data types are used to store collections of items. You can access, manipulate, and iterate through sequence data types, and each has its unique features

Strings: Strings are sequences of characters enclosed in single or double quotes. You can perform operations like concatenation, slicing, and formatting on strings.

Lists: Lists are ordered collections that can hold different data types. You can append, remove, and modify elements in a list.

Arrays: Arrays are data structures typically used for numerical data and are provided by libraries like 'numpy' and 'array.' They allow for efficient mathematical operations on large datasets.

Tuples: Tuples are similar to lists but are immutable, meaning their content cannot be changed once defined.

Dictionary: Dictionaries are collections of key-value pairs, allowing for efficient data retrieval based on keys.

Sets: Sets are unordered collections of unique elements, and they support operations like union, intersection, and difference.

Range: The 'range' type generates sequences of numbers and is commonly used in loops.

Loops and Conditional Statements: Loops, such as 'for' and 'while' loops, enable repetitive execution of code. Conditional statements, like 'if,' 'elif,' and 'else,' allow you to make decisions based on specific conditions.

Linear Regression: Linear regression is a statistical modeling technique implemented in Python to analyze the relationship between two or more variables. The 'scikit-learn' library provides a powerful tool for building linear regression models. Simple linear regression deals with one independent variable and one dependent variable, while multiple linear regression involves multiple independent variables. These models use mathematical equations to predict a target variable based on input data. Evaluation metrics like Mean Squared Error (MSE) and R-squared (R^2) help assess the model's accuracy.

**Program:**

Variable Creation and Basic Operators

# Variable Creation and Basic Operators

x = 5 # Create a variable 'x' and assign the value 5

y = 3

sum\_result = x + y # Addition

difference = x - y # Subtraction

product = x \* y # Multiplication

division\_result = x / y # Division

modulus\_result = x % y # Modulus

logical\_result = x > y # Logical comparison

Data Types and Associated Operations

# Data Types and Associated Operations

name = "Alice" # String

age = 30 # Integer

height = 1.75 # Float

boolean\_value = True # Boolean

# Type Casting

age\_str = str(age) # Convert integer to string

height\_int = int(height) # Convert float to integer

# String Operations

greeting = "Hello, " + name # String concatenation

Sequence Data Types and Associated Operations

# Lists

my\_list = [1, 2, 3, 4, 5]

my\_list.append(6) # Add an element to the list

sliced\_list = my\_list[1:4] # Slice a portion of the list

# Tuples

my\_tuple = (10, 20, 30)

value = my\_tuple[1] # Accessing a tuple element

# Arrays (Using NumPy)

import numpy as np

my\_array = np.array([1, 2, 3, 4, 5])

Dictionaries, Sets, and Range

# Dictionaries

my\_dict = {"name": "Bob", "age": 25}

my\_dict["city"] = "New York" # Add a key-value pair

# Sets

my\_set = {1, 2, 3, 4, 5}

union\_set = my\_set.union({4, 5, 6}) # Union of two sets

intersection\_set = my\_set.intersection({3, 4, 5}) # Intersection of two sets

# Range

my\_range = range(1, 6) # Create a range of numbers from 1 to 5

Loops and Conditional Statements

# Conditional Statements

if age >= 18:

print("You are an adult")

else:

print("You are a minor")

# Loops

for i in range(5):

print(i) # Print numbers from 0 to 4

# While Loop

counter = 0

while counter < 5:

print(counter)

counter += 1 # Increment counter

Introduction to Linear Regression (Using scikit-learn)

# Linear Regression

import numpy as np

from sklearn.linear\_model import LinearRegression

import matplotlib.pyplot as plt

# Sample Data

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)

y = np.array([2, 4, 5, 4, 5])

# Create and Train the Model

model = LinearRegression()

model.fit(X, y)

# Make Predictions

y\_pred = model.predict(X)

# Plot the Data and Regression Line

plt.scatter(X, y) # Scatter plot of data points

plt.plot(X, y\_pred, color='red') # Regression line

plt.show() # Display the plot

**Conclusion:**

This comprehensive experiment provides an introduction to a wide range of Python programming topics, along with the fundamentals of linear regression for data analysis.

**PRACTICAL-03**

**Aim:**

To learn different Libraries in Python and to perform Simple Linear Regression and Multiple Linear Regression

**Objective:**

In this experiment, you will learn how to perform Simple Linear Regression and Multiple Linear Regression using Python libraries. We will generate synthetic data and use various libraries for data manipulation, model training, and evaluation.

**Materials Required:**

1. `numpy`: For numerical operations and generating data.

2. `pandas`: For data manipulation and analysis.

3. `scikit-learn`: For building regression models.

**Theory:**

To delve into the world of machine learning and data analysis, it's crucial to learn different libraries in Python and apply them to tasks like Simple Linear Regression and Multiple Linear Regression. Python's extensive library ecosystem includes essential tools like 'numpy' for numerical computations and 'pandas' for data manipulation, making it a go-to language for data science. 'Scikit-learn' is a powerful library for machine learning that facilitates building regression models. Simple Linear Regression models the relationship between one independent variable and a dependent variable, while Multiple Linear Regression extends this to multiple independent variables. These models utilize mathematical equations to make predictions based on input data. Evaluating their accuracy is pivotal, and metrics like Mean Squared Error (MSE) and R-squared (R^2) provide insights into model performance. By learning these libraries and regression techniques, you gain the capability to extract valuable insights from data, make predictions, and solve real-world problems in fields such as finance, healthcare, and marketing. It's a skill set highly sought after in today's data-driven world.

**Procedure:**

1. Import the necessary libraries.
2. Generate synthetic data.
3. Create DataFrames to work with the data.
4. Split the data into training and testing sets.
5. Train and evaluate the regression models.

**Program:**

import numpy as np

import pandas as pd

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generate synthetic data for Simple Linear Regression

np.random.seed(0)

X\_simple = 2 \* np.random.rand(100, 1)

y\_simple = 4 + 3 \* X\_simple + np.random.randn(100, 1)

# Generate synthetic data for Multiple Linear Regression

X\_multiple = 2 \* np.random.rand(100, 2)

y\_multiple = 4 + 3 \* X\_multiple[:, 0] + 2 \* X\_multiple[:, 1] + np.random.randn(100, 1)

# Create DataFrames for Simple Linear Regression

df\_simple = pd.DataFrame({'X': X\_simple.flatten(), 'y': y\_simple.flatten()})

# Create DataFrames for Multiple Linear Regression

df\_multiple = pd.DataFrame({'X1': X\_multiple[:, 0], 'X2': X\_multiple[:, 1], 'y': y\_multiple.flatten()})

# Split data for Simple Linear Regression

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

# Split data for Multiple Linear Regression

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

# Create and train Simple Linear Regression model

simple\_reg = LinearRegression()

simple\_reg.fit(X\_train\_simple, y\_train\_simple)

y\_pred\_simple = simple\_reg.predict(X\_test\_simple)

# Create and train Multiple Linear Regression model

multiple\_reg = LinearRegression()

multiple\_reg.fit(X\_train\_multiple, y\_train\_multiple)

y\_pred\_multiple = multiple\_reg.predict(X\_test\_multiple)

# Evaluate the models

mse\_simple = mean\_squared\_error(y\_test\_simple, y\_pred\_simple)

r2\_simple = r2\_score(y\_test\_simple, y\_pred\_simple)

mse\_multiple = mean\_squared\_error(y\_test\_multiple, y\_pred\_multiple)

r2\_multiple = r2\_score(y\_test\_multiple, y\_pred\_multiple)

print(f"Simple Linear Regression MSE: {mse\_simple}")

print(f"Simple Linear Regression R^2: {r2\_simple}")

print(f"Multiple Linear Regression MSE: {mse\_multiple}")

print(f"Multiple Linear Regression R^2: {r2\_multiple}")

**Conclusion:**

This experiment helped us learn how to implement and evaluate regression models using Python libraries. It covers the basics of Simple and Multiple Linear Regression, making it a valuable exercise for understanding linear regression techniques.

**PRACTICAL-04**

**Aim:**

To perform Logistic Regression

**Objective:**

1. To apply logistic regression to a real dataset.

2. To gain an understanding of the logistic regression algorithm and its application.

3. To evaluate the model's performance and interpret the results.

**Materials Required:**

1. A dataset with binary classification labels (e.g., a dataset with two classes like spam/ham emails, disease/no disease, etc.).

2. Python environment with libraries such as NumPy, pandas, and scikit-learn for data manipulation and logistic regression implementation.

**Theory:**

Logistic Regression is a fundamental classification algorithm used to predict binary outcomes, making it a cornerstone of machine learning and statistics. It's particularly well-suited for scenarios where you need to answer questions like "yes" or "no," such as spam or not spam, disease or no disease. At its core is the sigmoid function, which transforms a linear combination of input features into a probability value between 0 and 1. The logistic function creates an S-shaped curve, which is essential for mapping real-valued inputs to probabilities. The log-odds (logit) of the probability of the positive class is modeled as a linear function of the input features and corresponding coefficients. The decision boundary, where the probability equals 0.5, distinguishes the two classes. This boundary can be linear or nonlinear, depending on the data and weights. Logistic Regression estimates coefficients using maximum likelihood, which seeks to find the best-fitting model to the observed data. It can be regularized to prevent overfitting. Evaluation metrics like accuracy, precision, recall, and F1-score are commonly used to assess the model's performance. Overall, Logistic Regression is an interpretable and widely applied classification method in various fields, underpinned by its simple yet powerful theoretical framework.

**Procedure:**

Step 1: Data Preparation

- Load and preprocess the dataset, ensuring it contains binary class labels (0 and 1).

- Split the dataset into training and testing sets.

Step 2: Implement Logistic Regression

- Import relevant libraries.

- Create a logistic regression model using scikit-learn or implement logistic regression from scratch.

- Train the model using the training data.

- Optionally, tune hyperparameters like regularization strength (C) and solver based on your dataset.

Step 3: Model Evaluation

- Use the trained model to predict outcomes on the testing data.

- Calculate performance metrics like accuracy, precision, recall, F1-score, and the confusion matrix to evaluate the model's performance.

**Program:**

# Import necessary libraries

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

# Load and preprocess the dataset

data = pd.read\_csv("your\_dataset.csv")

X = data.drop("target", axis=1)

y = data["target"]

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

# Create and train the logistic regression model

logistic\_model = LogisticRegression()

logistic\_model.fit(X\_train, y\_train)

# Make predictions on the test set

predictions = logistic\_model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, predictions)

precision = precision\_score(y\_test, predictions)

recall = recall\_score(y\_test, predictions)

f1 = f1\_score(y\_test, predictions)

conf\_matrix = confusion\_matrix(y\_test, predictions)

**Conclusion:**

In this experiment, we applied logistic regression to a binary classification problem. We trained the model, evaluated its performance using various metrics, and interpreted the results. Logistic regression is a powerful algorithm for binary classification, and its effectiveness depends on the quality of data and appropriate parameter tuning.

**PRACTICAL-05**

**Aim:**

To perform Linear Discriminate Analysis

**Objective:**

1. Understand the concept of LDA and its application in dimensionality reduction.

2. Implement LDA to transform a dataset and visualize the results.

3. Evaluate the effectiveness of LDA in improving class separability.

**Materials Required:**

1. Python environment with libraries: NumPy, Pandas, Matplotlib, and Scikit-Learn.

2. A dataset with multiple classes for analysis.

**Theory:**

Linear Discriminant Analysis (LDA) is a powerful statistical technique used for dimensionality reduction and feature extraction in the field of pattern recognition and machine learning. Its core objective is to transform high-dimensional data into a lower-dimensional space while maximizing the separation between different classes or groups within the data. LDA operates under several key assumptions, including the normal distribution of data and the identical covariance of classes or an approximation thereof. It relies on two fundamental scatter matrices: the within-class scatter matrix (Sw) that quantifies the spread of data within each class, and the between-class scatter matrix (Sb) that measures the differences between class means, effectively capturing the separation between classes. The core of LDA involves solving a generalized eigenvalue problem to find the eigenvectors and eigenvalues of the matrix Sw^(-1) \* Sb. These eigenvectors represent the directions in the feature space where the data varies the most, and the eigenvalues indicate their importance. By selecting the top eigenvectors and projecting the data onto this new subspace, LDA enables the transformation of data in a way that enhances class separability, making it an invaluable tool for applications such as face recognition, feature selection, and improving the performance of classification algorithms.

**Procedure:**

Step 1: Data Preprocessing

- Load and preprocess the dataset.

- Divide the dataset into features (X) and target labels (y).

Step 2: Standardize the Data

- Standardize the feature matrix (X) to have mean = 0 and standard deviation = 1.

Step 3: Compute Class Means and Scatter Matrices

- Calculate the mean of each class (mean vectors).

- Compute the within-class scatter matrix (Sw) and between-class scatter matrix (Sb).

Step 4: Solve Eigenvalue Problem

- Find the eigenvalues and eigenvectors of the matrix Sw^-1 \* Sb.

Step 5: Select Linear Discriminants

- Select the top k eigenvectors corresponding to the largest eigenvalues to form a transformation matrix (W).

Step 6: Project Data onto Lower-Dimensional Space

- Project the data onto the new subspace by multiplying X by W to obtain X\_lda.

Step 7: Visualization

- Visualize the reduced dataset to assess class separability.

Step 8: Evaluate Performance

- Train a classifier (e.g., Logistic Regression) on the transformed data.

- Evaluate the classification performance (accuracy, F1-score, etc.).

**Program:**

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

from sklearn.datasets import load\_iris

# Load the Iris dataset as an example

iris = load\_iris()

X = iris.data

y = iris.target

# Create an LDA model

lda = LinearDiscriminantAnalysis(n\_components=2) # You can specify the number of components

# Fit and transform the data

X\_lda = lda.fit\_transform(X, y)

# Visualize the transformed data

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

colors = ['navy', 'turquoise', 'darkorange']

lw = 2

for color, i, target\_name in zip(colors, [0, 1, 2], iris.target\_names):

plt.scatter(X\_lda[y == i, 0], X\_lda[y == i, 1], color=color, alpha=0.8, lw=lw, label=target\_name)

plt.legend(loc='best', shadow=False, scatterpoints=1)

plt.title('LDA of IRIS dataset')

plt.show()

This code demonstrates LDA on the Iris dataset, transforming it into a lower-dimensional space (in this case, 2D) and visualizing the results. We can replace the Iris dataset with our own dataset by loading our data into the `X` and `y` variables.

**Conclusion:**

In this experiment, we successfully applied Linear Discriminant Analysis to reduce the dimensionality of the dataset while maximizing class separability. The visualization of the reduced data showed improved class separation, and the classifier trained on the transformed data demonstrated enhanced classification performance. LDA can be a valuable tool for feature selection and dimensionality reduction in various machine learning tasks.

**PRACTICAL-06**

**Aim:**

To perform Quadratic Discriminate Analysis

**Objective:**

1. Understand the concept of QDA and its application in classification.

2. Implement QDA to classify data into multiple classes.

3. Evaluate the classification performance of QDA.

**Materials Required:**

1. Python environment with libraries: NumPy, Pandas, Matplotlib, and Scikit-Learn.

2. A dataset with multiple classes for analysis.

**Theory:**

Quadratic Discriminant Analysis (QDA) is a classification technique used in statistics and machine learning to separate data points into multiple classes. Unlike Linear Discriminant Analysis (LDA), which assumes that all classes share a common covariance matrix and forms linear decision boundaries, QDA is more flexible. QDA allows for varying covariance structures between classes, making it suitable for situations where different classes exhibit different levels of variation and complex relationships with the features.

In QDA, each class is modeled as a multivariate Gaussian distribution, characterized by its mean vector and covariance matrix. The method estimates the probability of a data point belonging to each class based on its feature values. Importantly, QDA models the decision boundaries as quadratic functions, permitting the capture of intricate, non-linear relationships between features and classes. This flexibility makes QDA valuable when linear boundaries are insufficient, and more complex decision boundaries are needed to accurately classify data points.

**Procedure:**

Step 1: Data Preprocessing

- Load and preprocess the dataset.

- Divide the dataset into features (X) and target labels (y).

Step 2: Train QDA Model

- Create a QDA model using Scikit-Learn.

Step 3: Model Training

- Train the QDA model on the dataset.

Step 4: Classification

- Use the trained QDA model to classify data points into different classes.

Step 5: Evaluation

- Evaluate the classification performance of QDA using metrics like accuracy, precision, recall, and F1-score.

**Program:**

from sklearn.discriminant\_analysis import QuadraticDiscriminantAnalysis

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

from sklearn.datasets import load\_iris

from sklearn.metrics import accuracy\_score

# Load a sample dataset (you can replace this with your own data)

data = load\_iris()

X = data.data

y = data.target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create a QDA model

qda = QuadraticDiscriminantAnalysis()

# Fit the model to the training data

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

# Make predictions on the test data

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

# Calculate the accuracy of the model

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy: {:.2f}%".format(accuracy \* 100))

**Conclusion:**

In this experiment, we successfully applied Quadratic Discriminant Analysis (QDA) to classify data points into multiple classes. QDA is valuable when the assumption of equal covariance matrices for classes is not valid. We evaluated the classification performance of the QDA model using standard metrics, providing insights into its effectiveness in multi-class classification tasks. QDA is a powerful technique when dealing with non-linear decision boundaries and varying covariance structures across classes.

**PRACTICAL-07**

**Aim:**

To implement K-Nearest Neighbors technique

**Objectives:**

1. Understand the K-NN algorithm and its principles.

2. Implement K-NN using Python and scikit-learn.

3. Evaluate the algorithm's performance on a real dataset.

**Materials Required:**

- Python environment (with libraries such as scikit-learn, NumPy, and Matplotlib)

- A dataset for classification

- Computer with sufficient processing power

**Theory:**

The K-Nearest Neighbors (K-NN) technique is a fundamental machine learning algorithm that operates on the principle of proximity. It is used for both classification and regression tasks. K-NN assigns a class or predicts a value for a data point based on the majority vote or average, respectively, of its K-nearest neighbors in the feature space. The choice of the distance metric, often Euclidean distance, allows the algorithm to measure the similarity between data points. A smaller K value results in a more sensitive model that may overfit to noise, while a larger K value leads to a more generalized model. In classification, K-NN calculates distances from the point to be classified to all data points in the training dataset and selects the K-nearest neighbors. The majority class among these neighbors becomes the predicted class. In regression, K-NN predicts the value based on the average or weighted average of the K-nearest neighbors' target values. K-NN is simple to understand and implement, making it suitable for small to moderately sized datasets. However, it can be computationally expensive for large datasets, and its performance is sensitive to the choice of K and the quality of data preprocessing.

**Procedure:**

1. Data Preparation:

- Load a dataset that contains features and corresponding labels.

- Split the dataset into training and testing sets.

2. K-NN Implementation:

- Import the necessary libraries (scikit-learn, NumPy).

- Create a K-NN classifier object with the desired number of neighbors (K).

- Fit the classifier to the training data.

3. Model Evaluation:

- Make predictions on the testing data.

- Calculate classification metrics such as accuracy, precision, recall, and F1-score.

- Visualize the results using a confusion matrix and/or other appropriate plots.

**Program:**

from sklearn.neighbors import KNeighborsClassifier

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

from sklearn.metrics import accuracy\_score

# Sample data (replace with your own dataset)

X = [[1, 2], [2, 3], [3, 4], [3, 2], [4, 3]]

y = [0, 0, 1, 1, 0] # Sample labels

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create a K-NN classifier

k = 3 # Number of neighbors

knn = KNeighborsClassifier(n\_neighbors=k)

# Fit the classifier to the training data

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

# Make predictions on the test data

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

# Calculate the accuracy of the model

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

**Conclusion:**

In this experiment, we successfully implemented the K-Nearest Neighbors (K-NN) algorithm to classify data points. The accuracy and other classification metrics can be used to assess the model's performance. K-NN is a versatile algorithm and can be further tuned by changing the number of neighbors (K) or using different distance metrics for better results.

**PRACTICAL-08**

**Aim:**

To learn and perform The Lasso Decision Trees

**Objective:**

To understand and implement two distinct machine learning techniques, the Lasso regression and Decision Trees, and compare their performance for a regression task

**Materials Required:**

1. Python environment (e.g., Jupyter Notebook)

2. Python libraries: NumPy, pandas, scikit-learn

3. Dataset for regression (e.g., a CSV file containing features and target variable)

4. Computer with sufficient computational resources

**Theory:**

Lasso Decision Trees, also known as Decision Trees with L1 regularization, combine the interpretability of traditional Decision Trees with the feature selection capabilities of Lasso regression. This technique is particularly useful for handling high-dimensional datasets and reducing overfitting.

Lasso Decision Trees aim to create a decision tree that not only predicts target values but also selects a subset of relevant features by applying L1 regularization. During the learning phase, the algorithm recursively partitions the dataset based on the feature that offers the best split, similar to traditional Decision Trees. However, in the case of Lasso Decision Trees, it simultaneously computes the feature importance scores using L1 regularization, favoring features that contribute more to reducing the error.

This dual objective of splitting the data for accurate predictions and identifying feature importance is achieved through optimization techniques like coordinate descent. The regularization term encourages some feature coefficients to be exactly zero, effectively excluding them from the decision tree. As a result, the final tree structure is pruned, and only a subset of features is retained, making the model more interpretable and potentially less prone to overfitting in high-dimensional data.

**Procedure:**

1. Load the dataset: Import the necessary libraries and load the dataset.

2. Data Preprocessing: Preprocess the dataset if required by handling missing values and encoding categorical features.

3. Data Splitting: Split the dataset into a training set and a test set (e.g., 80% training, 20% test).

4. Lasso Regression:

a. Train a Lasso Regression model on the training set.

b. Tune the alpha hyperparameter to control the level of regularization.

c. Make predictions on the test set.

5. Decision Trees:

a. Train a Decision Tree model on the training set.

b. Set hyperparameters like the maximum depth, minimum samples per leaf, and criterion.

c. Make predictions on the test set.

6. Model Evaluation:

a. Evaluate the performance of both the Lasso Regression and Decision Trees on the test set using appropriate regression metrics (e.g., Mean Squared Error, R-squared).

**Program:**

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import Lasso

from sklearn.tree import DecisionTreeRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

# Load the dataset

data = pd.read\_csv('your\_dataset.csv')

# Data splitting

X = data.drop('target\_variable', axis=1)

y = data['target\_variable']

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

# Lasso Regression

lasso\_model = Lasso(alpha=0.01) # Experiment with different alpha values

lasso\_model.fit(X\_train, y\_train)

lasso\_predictions = lasso\_model.predict(X\_test)

# Decision Trees

tree\_model = DecisionTreeRegressor(max\_depth=5) # Experiment with depth and other hyperparameters

tree\_model.fit(X\_train, y\_train)

tree\_predictions = tree\_model.predict(X\_test)

# Model evaluation

lasso\_mse = mean\_squared\_error(y\_test, lasso\_predictions)

lasso\_r2 = r2\_score(y\_test, lasso\_predictions)

tree\_mse = mean\_squared\_error(y\_test, tree\_predictions)

tree\_r2 = r2\_score(y\_test, tree\_predictions)

print("Lasso Regression - MSE:", lasso\_mse, "R-squared:", lasso\_r2)

print("Decision Trees - MSE:", tree\_mse, "R-squared:", tree\_r2)

**Conclusion:**

In this experiment, we learned and implemented two distinct machine learning techniques: Lasso Regression and Decision Trees for regression tasks. By comparing their performance on a test dataset, we gain insights into the trade-offs between linear regression with feature selection (Lasso) and non-linear, tree-based modeling (Decision Trees). The choice between these techniques may depend on the nature of the dataset and the specific problem at hand.

**PRACTICAL-09**

**Aim:**

To learn and perform Fitting Classification Trees

**Objective:**

To learn and perform the construction of Classification Trees for predictive modeling. The objective is to understand the theory behind Classification Trees, implement the procedure using a dataset, and evaluate the tree's performance in making accurate predictions.

**Materials Required:**

1. Computer with Python and necessary libraries (e.g., scikit-learn, pandas)

2. Dataset for classification (e.g., the Iris dataset)

3. Jupyter Notebook or any Python development environment

**Theory:**

Learning to fit Classification Trees is a fundamental aspect of machine learning. Classification Trees are a type of supervised learning model used for categorical or discrete target variables. The theory behind fitting Classification Trees involves a recursive process of splitting the dataset into subsets based on the most discriminative features. This division continues until a stopping criterion is met, typically when a certain depth is reached or further partitioning does not significantly improve classification. The construction of the tree is guided by criteria such as Gini impurity or entropy, which measure the impurity or disorder in the dataset. Ultimately, a tree structure is formed, with leaf nodes representing class labels. This technique provides a visually interpretable model for decision-making and can be further optimized by adjusting hyperparameters like tree depth or minimum samples per leaf. Learning to fit Classification Trees is a crucial skill for building accurate and interpretable classification models in a wide range of applications.

**Procedure:**

1. Data Preparation:

- Load the dataset, e.g., the Iris dataset, using pandas.

- Split the dataset into features (X) and the target variable (y).

2. Tree Construction:

- Import the necessary libraries, such as `DecisionTreeClassifier` from scikit-learn.

- Create an instance of the classifier with the desired parameters, like the splitting criterion and maximum depth.

- Fit the model to the training data using `fit(X\_train, y\_train)`.

3. Visualization:

- Visualize the constructed tree for better understanding using libraries like `graphviz`.

4. Evaluation:

- Predict the class labels for the test dataset using `predict(X\_test)`.

- Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

5. Tuning and Optimization:

- Experiment with different hyperparameters to optimize the tree, e.g., adjusting the maximum depth or minimum samples per leaf.

**Program:**

# Import necessary libraries

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# Load the dataset

data = pd.read\_csv('iris.csv') # Replace with your dataset

X = data.drop('target', axis=1)

y = data['target']

# 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)

# Create and train the Decision Tree classifier

clf = DecisionTreeClassifier(criterion='gini', max\_depth=None)

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

# Make predictions

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

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

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

print(f'Accuracy: {accuracy}\nClassification Report:\n{report}')

**Conclusion:**

In this experiment, we learned how to construct a Classification Tree for predictive modeling. The decision tree was built, evaluated, and optimized to improve its performance. The experiment's objective was to gain hands-on experience with the theory and practice of Classification Trees, making it a valuable tool for solving classification problems in machine learning.

**PRACTICAL-10**

**Aim:**

To learn and perform Fitting Regression Trees

**Objective:**

To empower individuals to create accurate predictive models, gain insights into data, maintain model interpretability, and apply machine learning effectively across different domains. By achieving these objectives, one can make informed decisions, solve real-world problems, and open doors to more advanced machine learning techniques.

**Materials Required:**

1. Python with scikit-learn library

2. A dataset for regression (e.g., housing prices, stock prices)

3. Jupyter Notebook or any Python development environment

**Theory:**

Fitting Regression Trees is a fundamental concept in machine learning and data analysis. Regression trees are a versatile tool used for predicting continuous values by dividing the data into increasingly homogenous subsets. The core idea behind regression trees is to recursively partition the data based on the features that provide the best separation. The tree structure comprises nodes, each representing a feature and a splitting criterion. The leaves of the tree contain the predicted values for the response variable.

One of the key hyperparameters of regression trees is the maximum depth of the tree, which controls the complexity of the model. A deeper tree may capture intricate patterns but is prone to overfitting, while a shallower tree may result in underfitting.

The tree-building process involves finding the best feature and split point at each node, typically using metrics like Mean Squared Error (MSE) or Gini impurity. This recursive partitioning continues until a stopping criterion is met, such as reaching a maximum depth or having a minimum number of samples in a node.

Regression trees are interpretable and intuitive, allowing us to understand the relationships between input features and the target variable. However, they can be sensitive to the specific training data and may require techniques like pruning to improve generalization.

**Procedure:**

1. We import the necessary libraries, including scikit-learn for decision tree regression.

2. We generate synthetic data as an example. You should replace this with your own dataset.

3. The data is split into training and testing sets using `train\_test\_split`.

4. We create a `DecisionTreeRegressor` model with a specified `max\_depth` and fit it to the training data.

5. Predictions are made on the test data.

6. The model's performance is evaluated using the Mean Squared Error (MSE).

7. We visualize the regression tree rules, showing how the tree makes decisions on splitting the data. Note that visualizing the tree requires the 'graphviz' library, which you may need to install separately.

**Program:**

# Import necessary libraries

from sklearn.tree import DecisionTreeRegressor

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

from sklearn.metrics import mean\_squared\_error

import numpy as np

# Generate synthetic data for demonstration purposes (replace with your dataset)

np.random.seed(0)

X = np.sort(5 \* np.random.rand(80, 1), axis=0)

y = np.sin(X).ravel()

y[::5] += 3 \* (0.5 - np.random.rand(16))

# 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)

# Create and fit the regression tree model

model = DecisionTreeRegressor(max\_depth=5) # You can adjust max\_depth as needed

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

# Make predictions on the test data

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

# Evaluate the model's performance

mse = mean\_squared\_error(y\_test, y\_pred)

print("Mean Squared Error:", mse)

# Visualize the regression tree (requires 'graphviz' library)

from sklearn.tree import export\_text

tree\_rules = export\_text(model, feature\_names=["X"])

print("Regression Tree Rules:")

print(tree\_rules)

**Conclusion:**

In this experiment, we learned how to implement regression trees for predictive modeling. We loaded and preprocessed a dataset, split it into training and testing sets, created a regression tree model, made predictions, and evaluated the model's performance using the Mean Squared Error (MSE). Regression trees are useful for modeling and predicting continuous values, making them a valuable tool in machine learning and data analysis.

**PRACTICAL-11**

**Aim:**

To learn and perform Support Vector Classifier

**Objective:**

1. Learn the theory behind Support Vector Classifier (SVC).

2. Implement a Support Vector Classifier for a classification task.

3. Evaluate the model's performance using various metrics.

4. Explore the impact of different hyperparameters on SVC's performance.

5. Draw conclusions about the effectiveness of SVC in classification tasks.

**Materials Required:**

1. Python environment (e.g., Anaconda or Jupyter Notebook)

2. Scikit-Learn library

3. Real-world dataset for classification

4. Computer with sufficient processing power

**Theory:**

The Support Vector Classifier (SVC) is a supervised machine learning algorithm that specializes in binary classification. It operates by identifying an optimal hyperplane within the feature space that best separates data points belonging to different classes while maximizing the margin between them. This hyperplane is positioned to be equidistant from the nearest data points of each class, which are referred to as "support vectors." The margin, or the gap between these support vectors and the hyperplane, is a critical factor in SVC as it influences the model's generalization and ability to handle new data points.

SVC is versatile and is not limited to linear separation; it can be adapted to non-linear tasks through the use of kernel functions, such as polynomial, radial basis function (RBF), or sigmoid kernels. These kernels transform the original feature space into a higher-dimensional space where non-linear relationships can be better modeled. This enables SVC to tackle complex classification tasks effectively.

The primary goal of SVC is to find the hyperplane that achieves the maximum margin while still correctly classifying all training data points. When the data is not linearly separable, SVC aims to find the hyperplane that minimizes classification errors by introducing a trade-off between maximizing the margin and minimizing misclassification. This trade-off is controlled by a regularization parameter, usually denoted as C.

**Procedure:**

1. Import necessary libraries.

2. Load a real-world dataset for classification.

3. Preprocess the data, including feature scaling, encoding, and splitting it into training and testing sets.

4. Implement a Support Vector Classifier.

5. Train the SVC model on the training data.

6. Make predictions on the testing data.

7. Evaluate the model's performance using metrics such as accuracy, precision, recall, F1-score, and a confusion matrix.

8. Experiment with different hyperparameters, like the choice of kernel function and regularization parameters.

9. Analyze the impact of these hyperparameters on the model's performance.

10. Draw conclusions about the effectiveness and limitations of the Support Vector Classifier.

**Program:**

# Import necessary libraries

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.svm import SVC

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

# Load a real-world dataset (replace with your dataset)

# Example using Iris dataset for demonstration purposes

iris = load\_iris()

X, y = iris.data, iris.target

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create a Support Vector Classifier

svc\_classifier = SVC(kernel='linear', C=1)

# Train the SVC classifier

svc\_classifier.fit(X\_train, y\_train)

# Make predictions with SVC

svc\_predictions = svc\_classifier.predict(X\_test)

# Evaluate the SVC classifier

svc\_accuracy = accuracy\_score(y\_test, svc\_predictions)

print("SVC Accuracy:", svc\_accuracy)

print("Classification Report for SVC:")

print(classification\_report(y\_test, svc\_predictions))

print("Confusion Matrix for SVC:")

print(confusion\_matrix(y\_test, svc\_predictions))

**Conclusion:**

In this experiment, we learned about the Support Vector Classifier (SVC) and implemented it for a classification task. By training the model, making predictions, and evaluating its performance, we gained insights into how SVC can effectively classify data. We also explored the impact of different hyperparameters on the model's performance, providing a deeper understanding of how to fine-tune SVC for specific tasks.

**PRACTICAL-12**

**Aim:**

To learn and perform Support Vector Machine

**Objective:**

1. Learn the theory behind Support Vector Machine (SVM).

2. Implement a Support Vector Machine for a classification task.

3. Evaluate the model's performance using various metrics.

4. Explore the impact of different hyperparameters on SVM's performance.

5. Draw conclusions about the effectiveness of SVM in classification tasks.

**Materials Required:**

1. Python environment (e.g., Anaconda or Jupyter Notebook)

2. Scikit-Learn library

3. Real-world dataset for classification

4. Computer with sufficient processing power

**Theory:**

The Support Vector Machine (SVM) is a robust and versatile supervised machine learning algorithm used for classification and regression tasks. Its fundamental concept revolves around finding the optimal hyperplane that separates data points of different classes in a way that maximizes the margin between them. The "support vectors," which are the data points closest to the decision boundary, play a critical role in defining this hyperplane. SVM's primary objective is to identify a hyperplane that not only classifies the data accurately but also generalizes well to unseen data.

SVM's flexibility extends beyond linear separation; it can effectively handle non-linear classification tasks by employing kernel functions. These kernels transform the original feature space into a higher-dimensional space, where complex relationships between data points can be captured. Common kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid kernels. The choice of the kernel and other hyperparameters like the regularization parameter (C) allows SVM to adapt to a wide range of data patterns.

SVM's success is rooted in its ability to maximize the margin between classes, which translates into better generalization to new data. This robustness to overfitting and capacity to handle high-dimensional data make SVM a popular choice in various fields, including image classification, text categorization, and bioinformatics. Despite its power, SVM may be computationally intensive for large datasets, and the choice of the appropriate kernel and hyperparameters often requires fine-tuning to achieve optimal performance.

**Procedure:**

1. Import necessary libraries.

2. Load a real-world dataset for classification.

3. Preprocess the data, including feature scaling, encoding, and splitting it into training and testing sets.

4. Implement a Support Vector Machine.

5. Train the SVM model on the training data.

6. Make predictions on the testing data.

7. Evaluate the model's performance using metrics such as accuracy, precision, recall, F1-score, and a confusion matrix.

8. Experiment with different hyperparameters, like the choice of kernel function and regularization parameter (C).

9. Analyze the impact of these hyperparameters on the model's performance.

10. Draw conclusions about the effectiveness and limitations of the Support Vector Machine.

**Program:**

# Import necessary libraries

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.svm import SVC

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

# Load a real-world dataset (replace with your dataset)

# Example using Iris dataset for demonstration purposes

iris = load\_iris()

X, y = iris.data, iris.target

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create a Support Vector Machine

svm\_classifier = SVC(kernel='linear', C=1)

# Train the SVM classifier

svm\_classifier.fit(X\_train, y\_train)

# Make predictions with SVM

svm\_predictions = svm\_classifier.predict(X\_test)

# Evaluate the SVM classifier

svm\_accuracy = accuracy\_score(y\_test, svm\_predictions)

print("SVM Accuracy:", svm\_accuracy)

print("Classification Report for SVM:")

print(classification\_report(y\_test, svm\_predictions))

print("Confusion Matrix for SVM:")

print(confusion\_matrix(y\_test, svm\_predictions))

**Conclusion:**

In this experiment, we learned about the Support Vector Machine (SVM) and implemented it for a classification task. By training the model, making predictions, and evaluating its performance, we gained insights into how SVM can effectively classify data. We also explored the impact of different hyperparameters on the model's performance, providing a deeper understanding of how to fine-tune SVM for specific tasks.