**LAB MANUAL**

**FOR**

**Machine Learning (CSL-604)**

**(SEM - VI)**

**Artificial Intelligence & Data Science**

**FH-2025**

SUBJECT TITLE: MACHINE LEARNING

CLASS: T.E. AI & DS SEM: VI

**Lab Objectives**

1. To introduce platforms such as Anaconda, COLAB suitable to Machine learning.
2. To implement various Regression techniques.
3. To develop Neural Network based learning models.
4. To implement Clustering techniques.

**Lab Outcomes**

After successful completion of the course students will be able to:

1. Implement various Machine learning models.
2. Apply suitable Machine learning models for a given problem.
3. Implement Neural Network based models.
4. Apply Dimensionality Reduction techniques.

**LO-PO-PSO Mapping**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **LO vs (PO, PSO) Mapping** | | | | | | | | | | | | | | |
| **PO** | | | | | | | | | | | | | **PSO** | |
| **CO** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** |
| CSL604.1 | 2 | 2 | 1 | 1 | 2 | - | - | - | - | - | - | - | 2 | 2 |
| CSL604.2 | 1 | 2 | 2 | 3 | 3 | - | - | - | - | - | - | - | 2 | 2 |
| CSL604.3 | 2 | 2 | 2 | 2 | 2 | - | - | - | - | - | - | - | 3 | 3 |
| CSL604.4 | 1 | 2 | 2 | 2 | 2 | - | - | - | - | - | - | - | 1 | 1 |
| **AVG** | **1.5** | **2** | **1.75** | **2** | **2.25** | **-** | **-** | **-** | **-** | **-** | **-** | **-** | **2** | **2** |

**SUBJECT TITLE: MACHINE LEARNING**

CLASS: T.E. AI & DS SEM: VI

**List of Experiments**

| **Sr. No.** | **Name of Experiment** | **BL** | **Lab Outcomes** |
| --- | --- | --- | --- |
| 1 | Introduction to platforms such as Anaconda, COLAB | L-2 | LO1 |
| 2 | Study of Machine Learning Libraries and tools (Python library, tensor flow, keras) | L-2 | LO1 |
| 3 | To implement Linear Regression | L-3 | LO2 |
| 4 | To implement Logistic Regression | L-3 | LO2 |
| 5 | To implement Support Vector Machine | L-3 | LO2 |
| 6 | To implement Hebbian Learning | L-3 | LO3 |
| 7 | To implement Single Layer Perceptron Learning algorithm | L-3 | LO3 |
| 8 | To implement Error Backpropagation Perceptron Training Algorithm | L-3 | LO3 |
| 9 | To implement Principal Component Analysis | L-4 | LO4 |
| 10 | Applications of above algorithms as a case study (E. g. Handwriting Recognition using MNIST data set, classification using IRIS data set, etc) | L-6 | LO2 & LO4 |
| **Additional Experiments:** | | | |
| 1 | Design and implement an Expert system for classification useful for real world application. | L-6 | LO1&LO2 |
| 2 | Develop a regression model for prediction or forecasting, useful for real world application. | L-6 | LO1&LO2 |

Dr. Anagha S. Dhavalikar

**Subject In-charge**

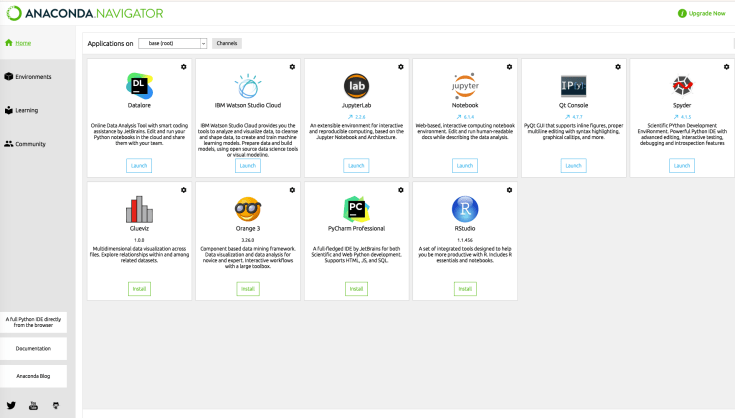
EXPERIMENT NO. 1

**Aim:** Introduction to platforms such as Anaconda, COLAB

**Prior Concepts:**

**Ref:** https://datatelier.com/anaconda-vs-google-colab-comparing-free-data-science-environments/

1. **Anaconda:**

****

Anaconda is an open-source platform designed for data science and machine learning. It provides a complete data science ecosystem, including Python and R programming languages, a package management system, and pre-installed libraries and tools for data manipulation, visualization, and analysis.

It comes bundled with a variety of popular data science libraries such as NumPy, Pandas, Matplotlib, and many more. With Anaconda, you can easily install, manage and switch between different libraries and packages, without worrying about compatibility issues.

In addition to the included libraries, Anaconda also includes popular IDEs such as Spyder and Jupyter Notebook. This means that developers and data scientists can create and manage their projects within the platform itself, making it a one-stop-shop for all their data science needs.

**Link for Anaconda:** [https://anaconda.org](https://anaconda.org/)

1. **Google Colab:**

Link: <https://colab.research.google.com/>

Google Colab, short for Google Colaboratory, is a cloud-based platform for data science and machine learning. In simple terms, Google Colab is a free online platform for developing and running machine learning models, algorithms, and code. It provides a Jupyter notebook environment, allowing you to write and execute Python code in a browser-based interface.

 One of the biggest advantages of Google Colab is that it’s fully cloud-based, meaning you don’t need to install anything on your computer to get started. All you need is a web browser and a Google account. This makes it a great choice for developers and data scientists who want to collaborate on projects or work remotely.

 Google Colab also provides access to powerful GPU resources, which can help speed up your machine learning projects significantly. This is a major advantage over local development environments, which often lack the necessary hardware resources for training complex machine learning models.

 In addition, Google Colab provides built-in access to a variety of popular libraries and tools, including TensorFlow, Keras, and PyTorch. This means you can get started with machine learning right away, without having to worry about installing and configuring different libraries and dependencies.

**Link for Google Colab:** https://colab.research.google.com/

**Learning Objectives:**

To understand platforms such as Anaconda and COLAB.

**Conclusion/Learning Outcomes:**

Both Anaconda and Google Colab have their own unique advantages and disadvantages. Choosing between them depends on the specific needs and preferences. If we need more control over the development environment and are working on sensitive data, Anaconda may be the better choice. On the other hand, if we need a free, cloud-based platform for collaboration and convenience, Google Colab can be useful.

EXPERIMENT NO. 2

**Aim:** Study of Machine Learning Libraries and tools (Python library, tensor flow, keras, etc.)

**Prior Concepts:**

**Ref:** <https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/>

Machine Learning is the science of programming a computer by which they are able to learn from different kinds of data. A more general definition given by Arthur Samuel is – “Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.” They are typically used to solve various types of life problems.

In the older days, people used to perform Machine Learning tasks by manually coding all the algorithms and mathematical and statistical formulas. This made the processing time-consuming, tedious, and inefficient. But in the modern days, it has become very much easy and more efficient compared to the olden days with various python libraries, frameworks, and modules. Today, Python is one of the most popular programming languages for this task and it has replaced many languages in the industry, one of the reasons is its vast collection of libraries. Python libraries that are used in Machine Learning are:

* Numpy
* Scipy
* Scikit-learn
* Theano
* TensorFlow
* Keras
* PyTorch
* Pandas
* Matplotlib

1. **NumPy:**

NumPy is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions. It is very useful for fundamental scientific computations in Machine Learning. It is particularly useful for linear algebra, Fourier transform, and random number capabilities. High-end libraries like TensorFlow use NumPy internally for manipulation of Tensors.

1. **Scipy:**

SciPy is a very popular library among Machine Learning enthusiasts as it contains different modules for optimization, linear algebra, integration and statistics. There is a difference between the SciPy library and the SciPy stack. The SciPy is one of the core packages that make up the SciPy stack. SciPy is also very useful for image manipulation.

1. **Scikit-learn:**

Scikit-learn is one of the most popular ML libraries for classical ML algorithms. It is built on top of two basic Python libraries, viz., NumPy and SciPy. Scikit-learn support most of the supervised and unsupervised learning algorithms. Scikit-learn can also be used for data-mining and data-analysis, which makes it a great tool who is starting out with ML.

1. **Theano:**

Machine Learning is basically mathematics and statistics. Theano is a popular python library that is used to define, evaluate and optimize mathematical expressions involving multi-dimensional arrays in an efficient manner. It is achieved by optimizing the utilization of CPU and GPU. It is extensively used for unit-testing and self-verification to detect and diagnose different types of errors. Theano is a very powerful library that has been used in large-scale computationally intensive scientific projects for a long time but is simple and approachable enough to be used by individuals for their own projects.

1. **TensorFlow:**

TensorFlow is a very popular open-source library for high performance numerical computation developed by the Google Brain team in Google. As the name suggests, Tensorflow is a framework that involves defining and running computations involving tensors. It can train and run deep neural networks that can be used to develop several AI applications. TensorFlow is widely used in the field of deep learning research and application.

1. **Keras:**

It provides many inbuilt methods for groping, combining and filtering data. Keras is a very popular Machine Learning library for Python. It is a high-level neural networks API capable of running on top of TensorFlow, CNTK, or Theano. It can run seamlessly on both CPU and GPU. Keras makes it really for ML beginners to build and design a Neural Network. One of the best thing about Keras is that it allows for easy and fast prototyping.

1. **PyTorch:**

PyTorch is a popular open-source Machine Learning library for Python based on Torch, which is an open-source Machine Learning library that is implemented in C with a wrapper in Lua. It has an extensive choice of tools and libraries that support Computer Vision, Natural Language Processing(NLP), and many more ML programs. It allows developers to perform computations on Tensors with GPU acceleration and also helps in creating computational graphs.

1. **Pandas:**

Pandas is a popular Python library for data analysis. It is not directly related to Machine Learning. As we know that the dataset must be prepared before training. In this case, Pandas comes handy as it was developed specifically for data extraction and preparation. It provides high-level data structures and wide variety tools for data analysis. It provides many inbuilt methods for grouping, combining and filtering data.

1. **Matplotlib:**

Matplotlib is a very popular Python library for data visualization. Like Pandas, it is not directly related to Machine Learning. It particularly comes in handy when a programmer wants to visualize the patterns in the data. It is a 2D plotting library used for creating 2D graphs and plots. A module named pyplot makes it easy for programmers for plotting as it provides features to control line styles, font properties, formatting axes, etc. It provides various kinds of graphs and plots for data visualization, viz., histogram, error charts, bar charts, etc.

 **Learning Objectives:**

To understand and explore Machine Learning Libraries and tools.

**Conclusion/Learning Outcome:**

Different types of Machine Learning Libraries and tools are explored and studied their applications in Machine Learning.

EXPERIMENT NO. 3

**Aim:** To implement Linear Regression.

**Prior Concepts:**

Linear Regression is an algorithm that belongs to supervised Machine Learning. It tries to apply relations that will predict the outcome of an event based on the independent variable data points. The relation is usually a straight line that best fits the different data points as close as possible. The output is of a continuous form, i.e., numerical value. For example, the output could be revenue or sales in currency, the number of products sold, etc. In the above example, the independent variable can be single or multiple.

### 

### Linear Regression Line

Linear regression can be expressed mathematically as:

y= β0+ β 1x+ ε

Here,

Y= Dependent Variable

X= Independent Variable

β 0= intercept of the line

β1 = Linear regression coefficient (slope of the line)

ε = random error

The last parameter, random error ε, is required as the best fit line also doesn't include the data points perfectly.

### **Linear Regression Model**

Since the Linear Regression algorithm represents a linear relationship between a dependent (y) and one or more independent (y) variables, it is known as Linear Regression. This means it finds how the value of the dependent variable changes according to the change in the value of

## Types of Linear Regression

Linear Regression can be broadly classified into two types of algorithms:

### **Simple Linear Regression**

A simple straight-line equation involving slope (dy/dx) and intercept (an integer/continuous value) is utilized in simple Linear Regression. Here a simple form is y=mx+c where y denotes the output x is the independent variable, and c is the intercept when x=0. With this equation, the algorithm trains the model of machine learning and gives the most accurate output

### **Multiple Linear Regression**

When a number of independent variables more than one, the governing linear equation applicable to regression takes a different form like: y= c+m1x1+m2x2… mnxn

where represents the coefficient responsible for impact of different independent variables x1, x2 etc. This machine learning algorithm, when applied, finds the values of coefficients m1, m2, etc., and gives the best fitting line.

### **Non-Linear Regression**

When the best fitting line is not a straight line but a curve, it is referred to as Non-Linear Regression.   **Algorithm:**

i. Use a LoadDataSet() function to open text file with tab delimited values and assume

the last value is the target value.

ii.Use second function, standRegres(), to compute the best-fit line as follows:

Load the x and y arrays and then convert them into matrices.

Compute XTX and then test if its determinate is zero .

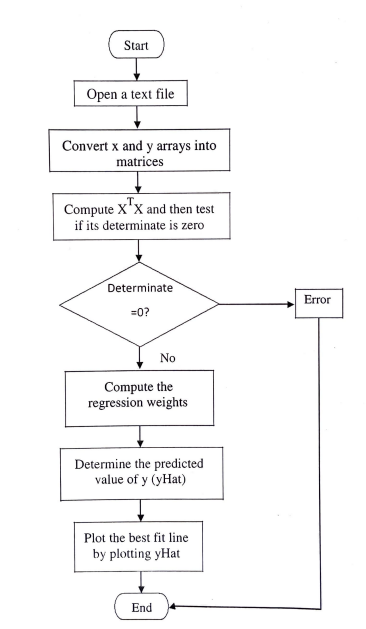
if the determinate is zero then you’ll get error

if the determinate is nonzero, you compute regression weights, ws and

return them

iii. Using ws, determinate the predicted value of y, by multiplying the x Matrix and ws.

iv. sort the point in ascending order, and plot the best fit line by plotting y.

**Flowchart:**

1. **Python Code:**

# Import necessary libraries

import numpy as np

from sklearn.linear\_model import LinearRegression

import matplotlib.pyplot as plt

# Sample data

np.random.seed(0)

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

y = 3 \* X + 1 + np.random.randn(100, 1)  # y = 3X + 1 + some random noise

# Create a linear regression model

model = LinearRegression()

# Fit the model to the data

model.fit(X, y)

# Make predictions

X\_new = np.array([[0], [2]])  # New data points for prediction

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

# Plotting the data and the regression line

plt.scatter(X, y, label='Original data')

plt.plot(X\_new, y\_pred, 'r-', label='Regression line')

plt.xlabel('X')

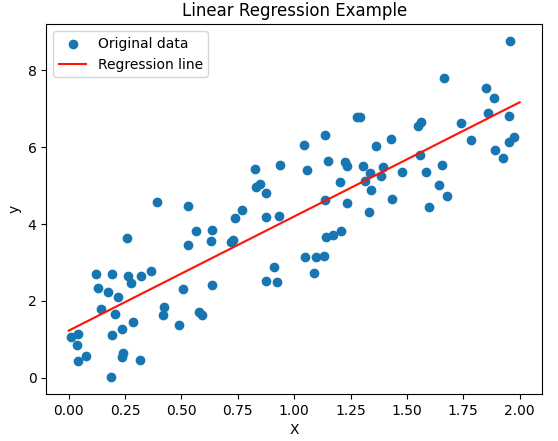
plt.ylabel('y')

plt.title('Linear Regression Example')

plt.legend()

plt.show()

**Output:**

****

1. **Python Code:**

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_regression

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error

# Generate a sample regression dataset

X, y = make\_regression(n\_samples=100, n\_features=1, noise=15, random\_state=42)

# 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 a Linear Regression model

model = LinearRegression()

# Fit the model to the training data

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

# Make predictions on the test set

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

# Calculate the coefficient of determination (R^2) and the mean squared error

r\_squared = model.score(X\_test, y\_test)

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

print(f"R-squared: {r\_squared}")

print(f"Mean Squared Error: {mse}")

# Plotting the results

plt.scatter(X\_test, y\_test, label='Actual data')

plt.plot(X\_test, y\_pred, color='red', label='Predicted line')

plt.xlabel('X')

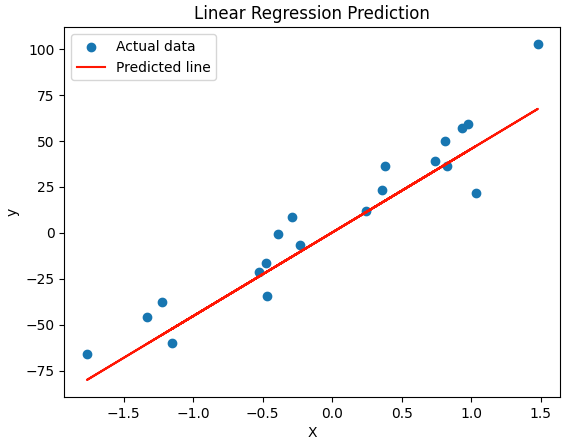
plt.ylabel('y')

plt.title('Linear Regression Prediction')

plt.legend()

plt.show()

**Output:**

****

**Learning Objectives:**

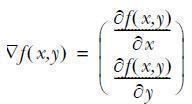
To understand and implement linear regression on sample dataset.

**Conclusion/Learning outcome:**

Regression analysis is a family of statistical tools that can help business analysts build models to predict trends, make tradeoff decisions, and model the real world for decision-making support. These models can be used to predict the value of one or more variables from knowledge of the value of other variables. Specific regression techniques include simple linear regression analysis, multiple linear regression analysis, multiple curvilinear regression, multivariate linear regression, and multivariate polynomial regression.

EXPERIMENT NO. 4

**Aim:** To implement Logistic Regression.



**Prior Concepts:**

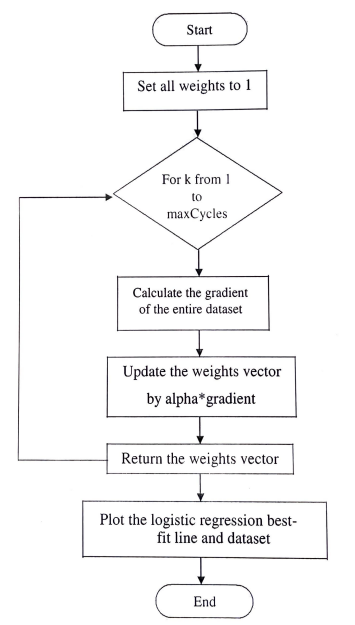
For the logistic regression classifier we’ll take our features and multiply each one by a weight and then add them up. This result will be put into the sigmoid, and we’ll get a number between 0 and 1. Anything above 0.5 we’ll classify as a 1, and anything below 0.5 we’ll classify as a 0. You can also think of logistic regression as a probability estimate. We will use optimization to find the best regression coefficients. The optimization algorithm used here is called gradient ascent. Gradient ascent is based on the idea that if we want to find the maximum point on a function, then the best way to move is in the direction of the gradient. We write the gradient with the symbol and the gradient of a function f(x,y) is given by the equation:

So this gradient means that we’ll move in the x direction by amount  and in the y direction by amount  . The function f(x,y) needs to be defined and differentiable around the points where it’s being evaluated.

# Algorithm:

* 1. Start with the weights all set to 1
  2. Repeat R number of times:
     + Calculate the gradient of the entire dataset
     + Update the weights vector by alpha\*gradient
     + Return the weights vector
  3. Plot the logistic regression best-fit line and dataset.

**FlowChart:**



**Python Code:**

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.linear\_model import LogisticRegression

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

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

# Load the Iris dataset

iris = load\_iris()

X = iris.data[:, :2]  # Considering only the first two features for visualization

y = iris.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 a Logistic Regression model

model = LogisticRegression(max\_iter=1000)

# Fit the model to the training data

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

# Make predictions on the test set

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

# Calculate the accuracy of the model

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

print(f"Accuracy: {accuracy}")

# Print confusion matrix and classification report

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

# Plotting the decision boundary

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

Z = model.predict(np.c\_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Set1)

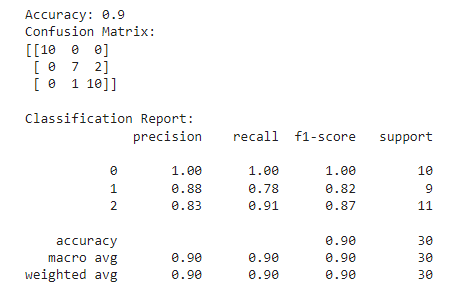
plt.xlabel('Sepal length')

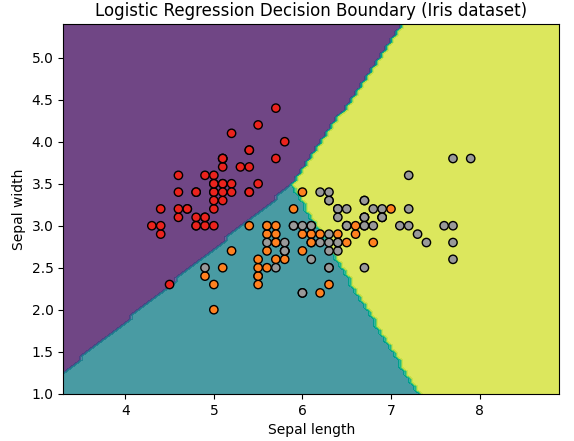
plt.ylabel('Sepal width')

plt.title('Logistic Regression Decision Boundary (Iris dataset)')

plt.show()

**Output:**

****

****

**Learning Objectives:**

To understand and implement logistic regression on dataset.

**Conclusion/Learning outcome:**

Logistic regression is a powerful tool, especially in epidemiologic studies, allowing multiple explanatory variables being analyzed simultaneously, meanwhile reducing the effect of confounding factors. However, researchers must pay attention to model building, avoiding just fee software with raw data and going forward to results. Some difficult decisions on model building v depend entirely on the expertise of researchers in the field.

EXPERIMENT NO. 5

**Aim:** To implement a Support Vector Machine.

**Prior Concept:**

In machine learning, support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. There are many implementations of support vector machines, but we’ll focus on one of the most popular implementations: the sequential minimal optimization (SMO) algorithm. SMO takes the large optimization problem and breaks it into many small problems. The small problems can easily be solved, and solving them sequentially will give you the same answer as trying to solve everything together. In addition to getting the same answer, the amount of time is greatly reduced.

**Algorithm:**

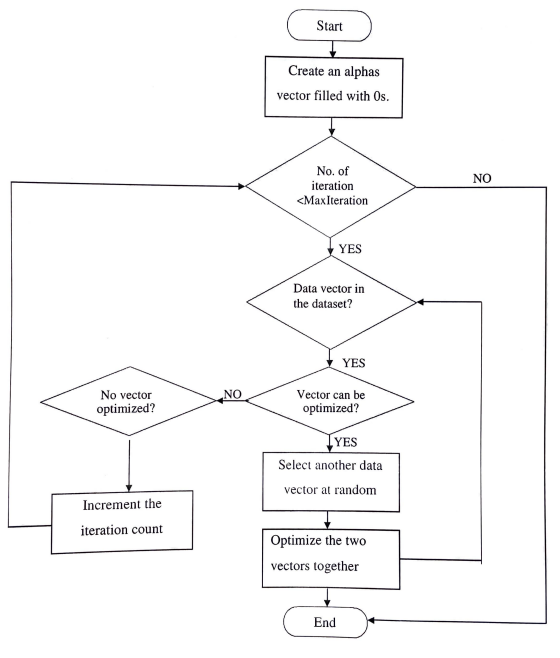
i. Create an alphas vector with 0s.

ii. While the number of iteration is less than Max Iterations:

1. For every data vector in the dataset
2. if the data vector can be optimized:
   1. Select another data vector at random
   2. Optimize the two vectors together
   3. if the vectors can’t be optimized then break

iii. If no vectors were optimized, then increment the iteration count

**Flow Chart:**



**Python Code:**

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.svm import SVC

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

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

# Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data[:, 2:]  # Using petal length and petal width

y = iris.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 an SVM classifier

svm\_classifier = SVC(kernel='linear', C=1.0, random\_state=42)

# Fit the model to the training data

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

# Make predictions on the test set

y\_pred = svm\_classifier.predict(X\_test)

# Calculate the accuracy of the model

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

print(f"Accuracy: {accuracy}")

# Print confusion matrix and classification report

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

# Plotting the decision boundary

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

Z = svm\_classifier.predict(np.c\_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Set1)

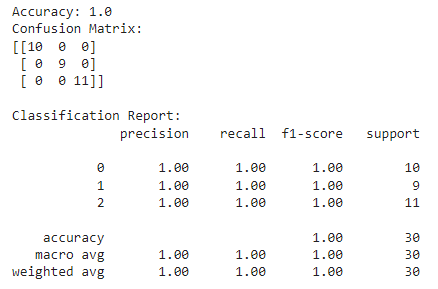
plt.xlabel('Petal length')

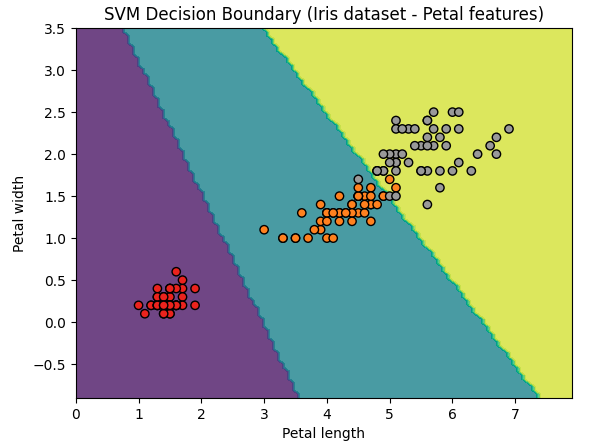
plt.ylabel('Petal width')

plt.title('SVM Decision Boundary (Iris dataset - Petal features)')

plt.show()

**Output:**

****

****

**Learning Objectives:**

To understand and implement Support Vector Machine.

**Conclusion/Learning outcome:**

SVM can be used to learn a variety of representations, such as neural nets, splines, polynomial estimators, etc, but there is a unique optimal solution for each choice of the SVM parameters, Support Vector Machines acts as one of the best approach to data modeling.

EXPERIMENT NO. 6

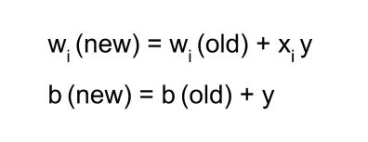
**Aim:** To implement Hebbian Learning Algorithm

**Prior Concept:**

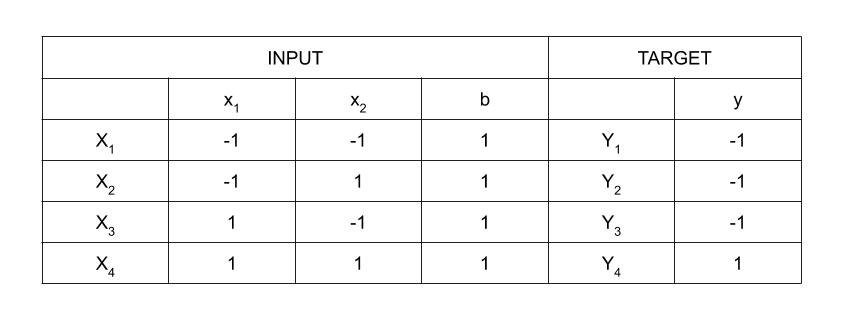
Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network. It is used for pattern classification. It is a single layer neural network, i.e. it has one input layer and one output layer. The input layer can have many units, say n. The output layer only has one unit. Hebbian rule works by updating the weights between neurons in the neural network for each training sample.

**Hebbian Learning Rule Algorithm:**

1. Set all weights to zero, wi= 0 for i=1 to n, and bias to zero.
2. For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.
3. Set activations for input units with the input vector Xi = Si for i = 1 to n.
4. Set the corresponding output value to the output neuron, i.e. y = t.
5. Update weight and bias by applying Hebb rule for all i = 1 to n:



**Implementing AND Gate:**



As shown in the Truth Table, there are 4 training samples, so there will be 4 iterations. Also, the activation function used here is Bipolar Sigmoidal Function so the range is [-1,1].

**Step 1 :**

Set weight and bias to zero, w = [ 0 0 0 ]Tand b = 0.

**Step 2 :**

Set input vector Xi = Sifor i = 1 to 4.

X1 = [ -1 -1 1 ]T

X2 = [ -1 1 1 ]T

X3 = [ 1 -1 1 ]T

X4 = [ 1 1 1 ]T

**Step 3 :**

Output value is set to y = t.

**Step 4 :**

Modifying weights using Hebbian Rule:

First iteration –

w(new) = w(old) + x1y1 = [ 0 0 0 ]T + [ -1 -1 1 ]T. [ -1 ] = [ 1 1 -1 ]T

For the second iteration, the final weight of the first one will be used and so on.

Second iteration –

w(new) = [ 1 1 -1 ]T + [ -1 1 1 ]T . [ -1 ] = [ 2 0 -2 ]T

Third iteration –

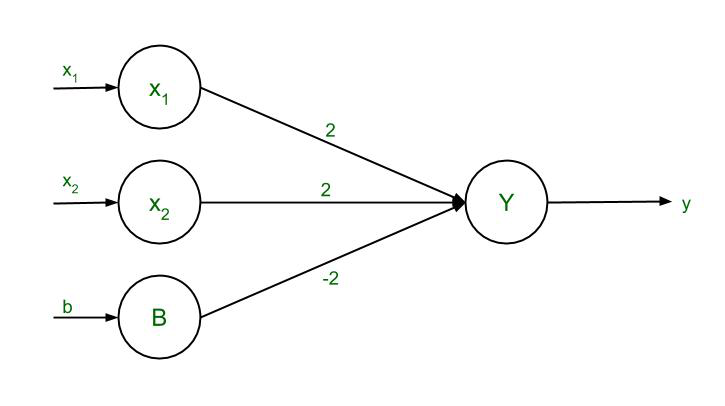
w(new) = [ 2 0 -2]T + [ 1 -1 1 ]T . [ -1 ] = [ 1 1 -3 ]T

Fourth iteration –

w(new) = [ 1 1 -3]T + [ 1 1 1 ]T . [ 1 ] = [ 2 2 -2 ]T

So, the final weight matrix is [ 2 2 -2 ]T

**Testing the network :**



*The network with the final weights*

For x1 = -1, x2 = -1, b = 1, Y = (-1)(2) + (-1)(2) + (1)(-2) = -6

For x1 = -1, x2 = 1, b = 1, Y = (-1)(2) + (1)(2) + (1)(-2) = -2

For x1 = 1, x2 = -1, b = 1, Y = (1)(2) + (-1)(2) + (1)(-2) = -2

For x1 = 1, x2 = 1, b = 1, Y = (1)(2) + (1)(2) + (1)(-2) = 2

The results are all compatible with the original table.

**Decision Boundary :**

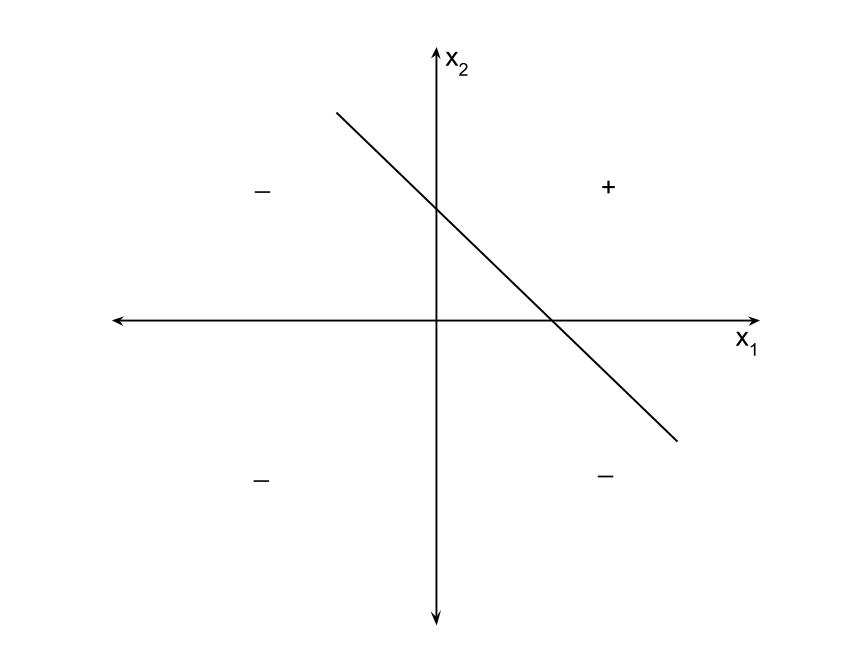
2x1 + 2x2 – 2b = y

Replacing y with 0, 2x1 + 2x2 – 2b = 0

Since bias, b = 1, so 2x1 + 2x2 – 2(1) = 0

2( x1 + x2 ) = 2

The final equation, x2 = -x1 + 1



*Decision Boundary of AND Function*

**Implementation of AND gate using Hebbian Learning Rule**

**Code:**

import numpy as np

# Input patterns for AND gate (bias term included)

X = np.array([

    [1, 1, 1],

    [1, 0, 0],

    [1, 1, 0],

    [1, 0, 1]

])

# Target outputs for AND gate

Y = np.array([1, 0, 0, 0])

# Initialize weights randomly

weights = np.random.rand(3)

# Learning rate

learning\_rate = 0.1

# Number of iterations (epochs)

epochs = 10

# Hebbian Learning Rule

for epoch in range(epochs):

    for i in range(len(X)):

        # Compute the output (activation) using current weights

        activation = np.dot(X[i], weights)

        # Update weights using Hebbian Learning Rule

        weights +=  learning\_rate \* activation \* X[i]

# Test the trained model

def predict(x):

    return 1 if np.dot(x, weights) > 0 else 0

# Test the AND gate logic

test\_inputs = np.array([

    [1, 1, 1],  # Should output 1 (True AND True)

    [0, 0, 0],  # Should output 0 (True AND False)

    [0, 1, 0],  # Should output 0 (True AND False)

    [0, 0, 1]   # Should output 0 (True AND False)

])

for i in range(len(test\_inputs)):

    prediction = predict(test\_inputs[i])

    print(f"Input: {test\_inputs[i][1:]} -> Output: {prediction}")

**Learning Objectives:**

To understand Hebbian Learning rule.

**Conclusion/Learning outcome:**

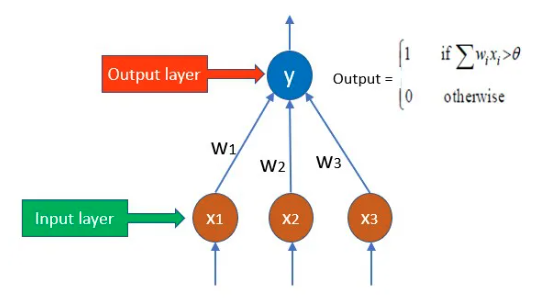
Hebbian Learning rule is understood and is used for implementing AND gate

EXPERIMENT NO. 7

**Aim:** To implement Single Layer Perceptron Learning algorithm

**Prior Concepts:**

In this article we will go through a single-layer perceptron this is the first and basic model of the artificial neural networks. It is also called the feed-forward neural network. The working of the single-layer perceptron (SLP) is based on the threshold transfer between the nodes. This is the simplest form of ANN and it is generally used in the linearly based cases for the machine learning problems.



**Algorithm:**

* In a single layer perceptron, the weights to each input node are assigned randomly since there is no a priori knowledge associated with the nodes.
* Also, a threshold value is assigned randomly.
* Now SLP sums all the weights which are inputted and if the sums are is above the threshold then the network is activated.
* If the calculated value is matched with the desired value, then the model is successful.
* If it is not, then since there is no back-propagation technique involved in this the error needs to be calculated using the below formula and the weights need to be adjusted again

#### **Perceptron Weight Adjustment**

Below is the equation in Perceptron weight adjustment:

**Δw = η \* d \* x**

**Where,**

* **d:** Predicted Output – Desired Output
* **η:** Learning Rate, Usually Less than 1.
* **x:** Input Data.

Since this network model works with the linear classification and if the data is not linearly separable, then this model will not show the proper results.

**Python Code:**

import numpy as np

# Define the input patterns for OR gate (including bias term)

X = np.array([

    [1, 1],

    [1, 0],

    [0, 1],

    [0, 0]

])

# Define the target outputs for OR gate

Y = np.array([1, 1, 1, 0])

# Initialize weights randomly (including bias weight)

weights = np.random.rand(3)

# Learning rate

learning\_rate = 0.1

# Number of epochs (iterations)

epochs = 10

# Perceptron Learning Algorithm

for epoch in range(epochs):

    for i in range(len(X)):

        # Add bias term to input

        x = np.insert(X[i], 0, 1)

        # Compute the activation (dot product of inputs and weights)

        activation = np.dot(x, weights)

        # Calculate the predicted output (1 if activation > 0, else 0)

        predicted\_output = 1 if activation > 0 else 0

        # Update weights based on the error

        error = Y[i] - predicted\_output

        weights += learning\_rate \* error \* x

# Function to predict output based on learned weights

def predict(x):

    # Add bias term to input

    x = np.insert(x, 0, 1)

    # Compute activation

    activation = np.dot(x, weights)

    # Return output (1 if activation is greater than 0, else 0)

    return 1 if activation > 0 else 0

# Test the trained model

test\_inputs = np.array([

    [1, 1],  # True OR True -> Should output 1

    [1, 0],  # True OR False -> Should output 1

    [0, 1],  # True OR False -> Should output 1

    [0, 0]   # False OR False -> Should output 0

])

print("Testing the trained model for OR gate logic:")

for i in range(len(test\_inputs)):

    prediction = predict(test\_inputs[i])

    print(f"Input: {test\_inputs[i]} -> Output: {prediction}")

**Output:**

Testing the trained model for OR gate logic:

Input: [1 1] -> Output: 1

Input: [1 0] -> Output: 1

Input: [0 1] -> Output: 1

Input: [0 0] -> Output: 0

**Learning Objectives:**

To understand and implement Single Layer Perceptron Learning algorithm.

**Conclusion/Learning outcome:**

Single Layer Perceptron Learning algorithm is understood and is used for implementing OR gate

EXPERIMENT NO. 8

**Aim:** To implement Error Back propagation Perceptron Training Algorithm

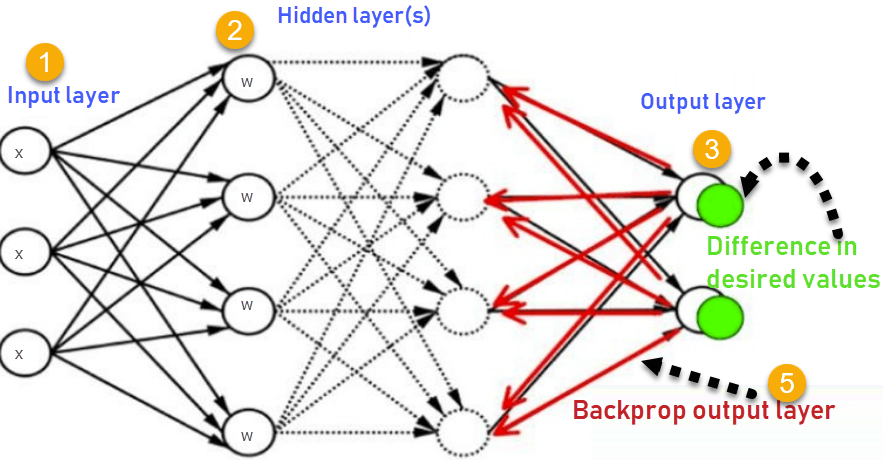
**Prior Concepts:**

Back propagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights reduces error rates and makes the model reliable by increasing its generalization. Back propagation in neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

## How Back propagation Algorithm Works

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule.

Consider the following Back propagation neural network example diagram to understand:



1. Inputs X, arrive through the pre-connected path
2. Input is modeled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

ErrorB= Actual Output – Desired Output

1. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved.

Most prominent advantages of Back propagation are:

* Back propagation is fast, simple and easy to program
* It has no parameters to tune apart from the numbers of input
* It is a flexible method as it does not require prior knowledge about the network
* It is a standard method that generally works well
* It does not need any special mention of the features of the function to be learned.

**Python Code:**

import numpy as np

#np.random.seed(0)

def sigmoid (x):

    return 1/(1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

#Input datasets

inputs = np.array([[0,0],[0,1],[1,0],[1,1]])

expected\_output = np.array([[0],[1],[1],[0]])

epochs = 10000

lr = 0.1

inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

#Random weights and bias initialization

hidden\_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))

hidden\_bias =np.random.uniform(size=(1,hiddenLayerNeurons))

output\_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))

output\_bias = np.random.uniform(size=(1,outputLayerNeurons))

print("Initial hidden weights: ",end='')

print(\*hidden\_weights)

print("Initial hidden biases: ",end='')

print(\*hidden\_bias)

print("Initial output weights: ",end='')

print(\*output\_weights)

print("Initial output biases: ",end='')

print(\*output\_bias)

#Training algorithm

for \_ in range(epochs):

  #Forward Propagation

  hidden\_layer\_activation = np.dot(inputs,hidden\_weights)

  hidden\_layer\_activation += hidden\_bias

  hidden\_layer\_output = sigmoid(hidden\_layer\_activation)

  output\_layer\_activation = np.dot(hidden\_layer\_output,output\_weights)

  output\_layer\_activation += output\_bias

  predicted\_output = sigmoid(output\_layer\_activation)

  #Backpropagation

  error = expected\_output - predicted\_output

  d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

  error\_hidden\_layer = d\_predicted\_output.dot(output\_weights.T)

  d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

  #Updating Weights and Biases

  output\_weights += hidden\_layer\_output.T.dot(d\_predicted\_output) \* lr

  output\_bias += np.sum(d\_predicted\_output,axis=0,keepdims=True) \* lr

  hidden\_weights += inputs.T.dot(d\_hidden\_layer) \* lr

  hidden\_bias += np.sum(d\_hidden\_layer,axis=0,keepdims=True) \* lr

print("Final hidden weights: ",end='')

print(\*hidden\_weights)

print("Final hidden bias: ",end='')

print(\*hidden\_bias)

print("Final output weights: ",end='')

print(\*output\_weights)

print("Final output bias: ",end='')

print(\*output\_bias)

print("\nOutput from neural network after 10,000 epochs: ",end='')

print(\*predicted\_output)

**Output:**

Initial hidden weights: [0.86055117 0.72704426] [0.27032791 0.1314828 ]

Initial hidden biases: [0.05537432 0.30159863]

Initial output weights: [0.26211815] [0.45614057]

Initial output biases: [0.68328134]

Final hidden weights: [3.40623101 5.55641764] [3.39232116 5.48326586]

Final hidden bias: [-5.18724528 -2.20651674]

Final output weights: [-7.35157564] [6.82917403]

Final output bias: [-3.06370489]

Output from neural network after 10,000 epochs: [0.08112647] [0.92197194] [0.9222861] [0.08599761]

**Learning Objectives:**

To understand and implement Error Back propagation Perceptron Training Algorithm.

**Conclusion/Learning outcome:**

An Error Back propagation Perceptron Training Algorithm is understood and used for implementing an XOR Gate.

EXPERIMENT NO. 9

**Aim:** To implement Principal Component Analysis.

**Theory:** Dimensionality reduction is the process of reducing the number of features (or dimensions) in a dataset while retaining as much information as possible. In other words, it is a process of transforming high-dimensional data into a lower-dimensional space that still preserves the essence of the original data. This can be done for a variety of reasons, such as to reduce the complexity of a model, to improve the performance of a learning algorithm, or to make it easier to visualize the data. There are several techniques for dimensionality reduction, including principal component analysis (PCA), singular value decomposition (SVD), and linear discriminant analysis (LDA). Each technique uses a different method to project the data onto a lower-dimensional space while preserving important information.

**Feature Selection:**

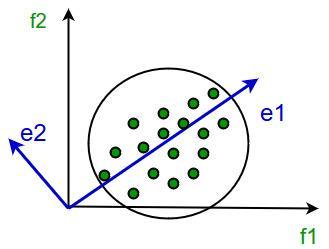
Feature selection involves selecting a subset of the original features that are most relevant to the problem at hand. The goal is to reduce the dimensionality of the dataset while retaining the most important features. There are several methods for feature selection, including filter methods, wrapper methods, and embedded methods. Filter methods rank the features based on their relevance to the target variable, wrapper methods use the model performance as the criteria for selecting features, and embedded methods combine feature selection with the model training process.

**Feature Extraction:**

Feature extraction involves creating new features by combining or transforming the original features. The goal is to create a set of features that captures the essence of the original data in a lower-dimensional space. There are several methods for feature extraction, including principal component analysis (PCA), linear discriminant analysis (LDA), and t-distributed stochastic neighbor embedding (t-SNE). PCA is a popular technique that projects the original features onto a lower-dimensional space while preserving as much of the variance as possible.

**Principal Component Analysis**

This method was introduced by Karl Pearson. It works on the condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.



It involves the following steps:

* Construct the covariance matrix of the data.
* Compute the eigenvectors of this matrix.
* Eigenvectors corresponding to the largest eigenvalues are used to reconstruct a large fraction of variance of the original data.

**Steps for PCA Algorithm**

1. **Standardize the data:** PCA requires standardized data, so the first step is to standardize the data to ensure that all variables have a mean of 0 and a standard deviation of 1.
2. **Calculate the covariance matrix:** The next step is to calculate the covariance matrix of the standardized data. This matrix shows how each variable is related to every other variable in the dataset.
3. **Calculate the eigenvectors and eigenvalues:** The eigenvectors and eigenvalues of the covariance matrix are then calculated. The eigenvectors represent the directions in which the data varies the most, while the eigenvalues represent the amount of variation along each eigenvector.
4. **Choose the principal components:** The principal components are the eigenvectors with the highest eigenvalues. These components represent the directions in which the data varies the most and are used to transform the original data into a lower-dimensional space.
5. **Transform the data:** The final step is to transform the original data into the lower-dimensional space defined by the principal components.

**Ref. Link:** <https://www.turing.com/kb/guide-to-principal-component-analysis>

**Code:**

from sklearn.decomposition import PCA

from sklearn.datasets import load\_iris

import pandas as pd

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Instantiate PCA and specify the number of components

pca = PCA(n\_components=2)

# Fit and transform the data to the new space

X\_pca = pca.fit\_transform(X)

# Create a DataFrame for visualization

df = pd.DataFrame(data=X\_pca, columns=['Principal Component 1', 'Principal Component 2'])

df['Target'] = y

# Display the transformed data

print(df.head())

**Output:**

Principal Component 1 Principal Component 2 Target

0 -2.684126 0.319397 0

1 -2.714142 -0.177001 0

2 -2.888991 -0.144949 0

3 -2.745343 -0.318299 0

4 -2.728717 0.326755 0

**Learning Objectives:**

To understand and implement Principal Component Analysis

**Conclusion/Learning outcome:**

Thus PCA algorithm is studied and implemented.

EXPERIMENT NO. 10

**Aim: C**ase study on HandWriting Recognition using MNIST data set

**OR**

**C**ase study on Classification using IRIS data set

**Theory:**

**Code:**

**Output:**

**Learning Objectives:**

**Conclusion/Learning outcome:**