

A.KAVITHA

2023 - 2024

4

Electronics and Communication Engineering

220801100

2116 220801100



24

23

CS 19411 Python Programming for Machine Learning

2116 220801100

ECE

IV

II

A KAVITHA

**Rajalakshmi Engineering College (Autonomous)**

**Chennai-602105**

|  |  |  |  |
| --- | --- | --- | --- |
| SI.No | Date | Name of the experiment | Signature |
| 1. | 15/2/2024 | Calculating values of random data using NumPy for mathematical formulas  1)Euclidean distance between two points 2) Dot Product of two Vectors 3)Solving a System of Linear Equations |  |
| 2. | 29/2/2024 | Write a simple Python code to generate random values and then compute their sigmoid and tanh (hyperbolic tangent) values using NumPy. Plot the values. |  |
| 3. | 7/3/2024 | simple Python program using pandas that creates a DataFrame, performs some basic operations, and  prints the result. |  |
| 4. | 14/3/2024 | Store and Load Excel / CSV files. |  |
| 5. | 21/3/2024 | Data Visualization |  |
| 6. | 11/4/2024 | Time Series |  |
| 7. | 25/04/024 | Linear regression model to predict the signal strength |  |
| 8. | 02/05/2024 | A component is defective or not based on Voltage and Current |  |
| 9. | 3/5/2024 | Decision tree classifier to predict signal quality based on transmitter, signal strength, and frequency |  |
| 10. | 09/5/2024 | k-NN classifier to predict signal quality based on distance from the transmitter, signal strength, and frequency |  |
| 11. | 16/5/2024 | Study of Artificial Neural Network (ANN) and Simple Program in ANN |  |
| 12. | 23/5/2024 | Study Of Support Vector Machine and and Simple Program in SVM |  |

**CS19411 Python Programming for Machine**

|  |  |
| --- | --- |
| **Exp no: 1** | **Calculating values of random data using NumPy for mathematical formulas 1)Euclidean distance between two points 2) Dot Product of two Vectors 3)Solving a System of Linear Equations** |

## AIM:

To calculate the values for the mathematical formulas using NumPy library

## INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) REQUIRED:

JUPYTER NOTEBOOK

## REQUIRED LIBRARIES FOR PYTHON:

* Numpy

## PROCEDURE:

1. **Euclidean distance**



## Dot Product



1. **Solving a System of Linear Equations**

A system of linear equations can be represented in matrix form as AX=B, whereA is the matrix of coefficients, X is the column vector of variables, and B is the column vector of solutions. To solve for **X,** we can use: X=A-1 B assuming A is invertible.

## PROGRAM:

**Calculating the Euclidean Distance Between Two Points**

import numpy as np

# Function to calculate Euclidean distance

def euclidean\_distance(point1, point2):

distance = np.sqrt(np.sum((point1 - point2) \*\* 2))

return distance

point1 = np.array([1, 2])

point2 = np.array([4, 6])

# Calculate distance

distance = euclidean\_distance(point1, point2)

print("Euclidean Distance:", distance)

**Calculating the Dot Product of Two Vectors**

import numpy as np

def dot\_product(vector1, vector2):

product = np.dot(vector1, vector2)

return product

vector1 = np.array([1, 2, 3])

vector2 = np.array([4, 5, 6])

# Calculate dot product

product = dot\_product(vector1, vector2)

print("Dot Product:", product)

**Solving a System of Linear Equations**

import numpy as np

def solve\_linear\_equations(A, B):

X = np.linalg.solve(A, B)

return X

# Example matrices

A = np.array([[3, 1], [1, 2]])

B = np.array([9, 8])

solution = solve\_linear\_equations(A, B)

print("Solution:", solution)

## Result:

Exercise 1 - Euclidean Distance: 5.

Exercise 2 - Dot Product: 32

Exercise 3 - Solution: [2. 3.]

|  |  |
| --- | --- |
| **Exp no: 2** | **Write a simple Python code to generate random values and then compute their sigmoid and tanh (hyperbolic tangent) values using NumPy. Plot the values.** |

## AIM:

To generate random values and to compute their sigmoid and tanh (hyperbolic tangent) values using NumPy. Plot the values.

**Program :**

import numpy as np

import matplotlib.pyplot as plt

# Generate random values

random\_values = np.random.randn(100)

# Compute sigmoid values

def sigmoid(x):

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

sigmoid\_values = sigmoid(random\_values)

# Compute tanh values

tanh\_values = np.tanh(random\_values)

# Plot the values

plt.figure(figsize=(12, 6))

# Plot sigmoid values

plt.subplot(1, 2, 1)

plt.plot(random\_values, sigmoid\_values, 'o', label='Sigmoid')

plt.title('Sigmoid Function')

plt.xlabel('Random Values')

plt.ylabel('Sigmoid Values')

plt.legend()

# Plot tanh values

plt.subplot(1, 2, 2)

plt.plot(random\_values, tanh\_values, 'o', label='Tanh', color='orange')

plt.title('Tanh Function')

plt.xlabel('Random Values')

plt.ylabel('Tanh Values')

plt.legend()

plt.tight\_layout()

plt.show()

**Result:**



|  |  |
| --- | --- |
| **Exp no: 3** | **simple Python program using pandas that creates a DataFrame, performs some basic operations, and prints the result.** |

**Aim:**

simple Python program using pandas that creates a DataFrame, performs some basic operations, and prints the result.

Steps:

1. Imports the pandas library as pd.
2. Creates two lists: data containing fruit names and prices containing their corresponding prices.
3. Zips these lists together and creates a DataFrame named fruits\_df with columns named "Fruit" and "Price".
4. Uses info() to get information about the DataFrame, including data types and number of entries.
5. Prints the entire DataFrame using to\_string().
6. Calculates descriptive statistics (mean, standard deviation, etc.) for the "Price" column and prints the results.

**Program Code:**

import pandas as pd

# Create a list of data

data = ["Apple", "Banana", "Cherry", "Orange", "Grape"]

prices = [1.25, 0.79, 2.00, 1.50, 0.99]

# Create a DataFrame

fruits\_df = pd.DataFrame(list(zip(data, prices)), columns = ['Fruit', 'Price'])

# Get basic information about the DataFrame

print(fruits\_df.info())

# Print the DataFrame

print(fruits\_df.to\_string())

# Get descriptive statistics of the 'Price' column

print(fruits\_df['Price'].describe())

**Result:**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 5 entries, 0 to 4

Data columns (total 2 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Fruit 5 non-null object

1 Price 5 non-null float64

dtypes: float64(1), object(1)

memory usage: 212.0+ bytes

None

Fruit Price

0 Apple 1.25

1 Banana 0.79

2 Cherry 2.00

3 Orange 1.50

4 Grape 0.99

count 5.000000

mean 1.306000

std 0.471307

min 0.790000

25% 0.990000

50% 1.250000

75% 1.500000

max 2.000000

Name: Price, dtype: float64

|  |  |
| --- | --- |
| **Exp no: 4** | **Store and Load Excel / CSV files.** |

**Aim:**

To store (save) and load data from Excel and CSV files using pandas.

Steps:

To Store:

import pandas as pd.

Create a sample DataFrame df.

Use to\_csv function to save the DataFrame to a CSV file.

* "people.csv" is the filename.
* index=True (default) saves the row index as a column. Set it to False to skip it.

To Load:

Import pandas as pd.

Use read\_csv to load data from a CSV file.

Use read\_excel to load data from an Excel file. By default, it reads the first sheet.

Specify the sheet name with the sheet\_name argument for loading data from a specific sheet.

**Program Code:**

**To Store:**

import pandas as pd

# Sample data

data = {"Name": ["Alice", "Bob", "Charlie"], "Age": [25, 30, 22]}

df = pd.DataFrame(data)

# Save to CSV file (with index)

df.to\_csv("people.csv", index=True)

# Save to CSV file (without index)

df.to\_csv("people\_no\_index.csv", index=False)

**To Load:**

import pandas as pd

# Load CSV data (assuming it has a header row)

df\_csv = pd.read\_csv("people.csv")

print(df\_csv)

**Result:**

Unnamed: 0 Name Age

0 0 Alice 25

1 1 Bob 30

2 2 Charlie 22

|  |  |
| --- | --- |
| **Exp no: 5** | **Data Visualization** |

**AIM:**

To Visulaize the give Data using Matplotlib.

**Program Code:**

import matplotlib.pyplot as plt

import pandas as pd # Optional for data manipulation

# Sample data (replace with your data or use pandas to read a CSV)

temperatures = [15, 18, 22, 20, 17, 24, 21, 19]

cities = ["New York", "Los Angeles", "Chicago", "Denver", "Seattle", "Miami", "Houston", "San Francisco"]

# Line plot

plt.plot(cities, temperatures, marker='o', linestyle='-') # Customize markers and line style

# Labels and title

plt.xlabel("City")

plt.ylabel("Temperature (°C)")

plt.title("Average Temperatures in Major US Cities")

# Display the plot

plt.xticks(rotation=45) # Rotate city names for better readability

plt.grid(True) # Add gridlines (optional)

plt.show()

**Result:**



|  |  |
| --- | --- |
| **Exp no: 6** | **Time Series** |

**AIM:**

To implement and check the time series function in the python.

**Program Code:**

import pandas as pd

import matplotlib.pyplot as plt

# Sample time series data (replace with your actual data)

data = {

"Date": pd.to\_datetime(["2023-01-01", "2023-02-01", "2023-03-01", "2023-04-01", "2023-05-01"]),

"Value": [100, 120, 135, 110, 145]

}

# Create DataFrame with Date as index

df = pd.DataFrame(data).set\_index("Date")

# Plot the time series

plt.figure(figsize=(10, 6)) # Adjust figure size for better viewing

plt.plot(df["Value"], marker='o', linestyle='-')

plt.xlabel("Date")

plt.ylabel("Value")

plt.title("Time Series Data")

plt.grid(True)

plt.show()

# Calculate daily change (optional)

df["Daily Change"] = df["Value"].diff() # Calculate difference between consecutive values

# Print descriptive statistics of daily change (optional)

print(df["Daily Change"].describe())

**Result:**



|  |  |
| --- | --- |
| **Exp no: 7** | **Linear regression model to predict the signal strength** |

**AIM:**

To develop a linear regression model to predict the signal strength based on the distance.

**Problem Statement**

We have a dataset that records the signal strength (in dBm) at various distances (in meters) from a transmitter. The goal is to develop a linear regression model to predict the signal strength based on the distance.

Steps: pip install numpy pandas scikit-learn matplotlib.

**Program Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn.linear\_model import LinearRegression

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

# Example dataset: Distance (meters) vs. Signal Strength (dBm)

data = {

'Distance': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

'Signal\_Strength': [-30, -35, -40, -45, -50, -55, -60, -65, -70, -75]

}

# Convert the data into a DataFrame

df = pd.DataFrame(data)

# Separate features and target variable

X = df[['Distance']].values # Feature: Distance

y = df['Signal\_Strength'].values # Target: Signal Strength

# 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 linear regression model

model = LinearRegression()

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

# Make predictions

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

# Evaluate the model

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

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

print(f'Mean Squared Error: {mse:.2f}')

print(f'R^2 Score: {r2:.2f}')

# Visualize the results

plt.scatter(X, y, color='blue', label='Actual Data')

plt.plot(X, model.predict(X), color='red', label='Fitted Line')

plt.xlabel('Distance (meters)')

plt.ylabel('Signal Strength (dBm)')

plt.title('Linear Regression: Distance vs. Signal Strength')

plt.legend()

plt.show()

**Result:**

Mean Squared Error: 0.00

R^2 Score: 1.00



|  |  |
| --- | --- |
| **Exp no: 8** | **A component is defective or not based on Voltage and Current** |

**Aim :**

To classify a component is defective or not based on Voltage and Current

**Program Code:**

import numpy as np

from sklearn.linear\_model import LogisticRegression

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

from sklearn.metrics import accuracy\_score

np.random.seed(0)

defective\_data = np.random.normal(loc=[5, 2], scale=[1, 0.5], size=(100, 2)) # Defective components

normal\_data = np.random.normal(loc=[8, 4], scale=[1, 0.5], size=(100, 2)) # Normal components

# Concatenate the data and create labels

X = np.concatenate([defective\_data, normal\_data])

y = np.concatenate([np.zeros(100), np.ones(100)]) # Defective: 0, Normal: 1

# Split the dataset 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 and train the logistic regression model

clf = LogisticRegression()

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

# Make predictions on the test set

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

# Calculate accuracy

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

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

**Result:**

Accuracy: 1.00

|  |  |
| --- | --- |
| **Exp no: 9** | **Decision tree classifier to predict signal quality based on transmitter, signal strength, and frequency** |

**Aim:**

create a simple dataset to classify signal quality based on various parameters such as distance from the transmitter, signal strength, and frequency.

**Problem Statement**:

Dataset that records various parameters affecting the signal quality (Good or Bad). The goal is to develop a decision tree classifier to predict signal quality based on these parameters.

**Steps:**

1. Dataset:
   * We create a simple dataset with distance from the transmitter, signal strength, frequency, and corresponding signal qua lity (Good or Bad). The dataset is stored in a dictionary and then converted into a pandas DataFrame.
2. Data Prepa ration:
   * Separate the dataset into features (X) and the target variable (y).
   * Encode the target variable Signal\_Quality from categorical values ('Good', 'Bad') to numerical values using LabelEncoder.
3. Model Training:
   * Split the data into training and testing sets using train\_test\_split.
   * Create an instance of DecisionTreeClassifier and train the model on the training data using the fit method.
4. Prediction and Evaluation:
   * Use the trained model to make predictions on the test data.
   * Calculate the accuracy score and generate a classification report to evaluate the model's performance.
5. Visualization:
   * Visualize the decision tree using plot\_tree to understand how the model makes decisions based on the input features.

**Program Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn.tree import DecisionTreeClassifier, plot\_tree

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

# Example dataset: Distance (meters), Signal Strength (dBm), Frequency (MHz) vs. Signal Quality

data = {

'Distance': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 3, 4, 5, 6],

'Signal\_Strength': [-30, -35, -40, -45, -50, -55, -60, -65, -70, -75, -33, -38, -43, -48, -53],

'Frequency': [850, 850, 850, 850, 850, 1900, 1900, 1900, 1900, 1900, 850, 850, 1900, 1900, 1900],

'Signal\_Quality': ['Good', 'Good', 'Good', 'Good', 'Bad', 'Bad', 'Bad', 'Bad', 'Bad', 'Bad', 'Good', 'Good', 'Bad', 'Bad', 'Bad']

}

# Convert the data into a DataFrame

df = pd.DataFrame(data)

# Separate features and target variable

X = df[['Distance', 'Signal\_Strength', 'Frequency']].values # Features

y = df['Signal\_Quality'].values # Target

# Encode the target variable

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y = le.fit\_transform(y) # 'Good' -> 1, 'Bad' -> 0

# 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

model = DecisionTreeClassifier(random\_state=42)

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

# Make predictions

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

# Evaluate the model

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

report = classification\_report(y\_test, y\_pred, target\_names=['Bad', 'Good'])

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

print('Classification Report:')

print(report)

# Visualize the decision tree

plt.figure(figsize=(20,10))

plot\_tree(model, feature\_names=['Distance', 'Signal\_Strength', 'Frequency'], class\_names=['Bad', 'Good'], filled=True)

plt.show()

**Output:**



**k-NN classifier to predict signal quality based on distance from the transmitter, signal strength, and frequency**

|  |  |
| --- | --- |
| **Exp no: 10** | **k-NN classifier to predict signal quality based on distance from the transmitter, signal strength, and frequency** |

**Aim:**

To classify signal quality based on various parameters such as distance from the transmitter, signal strength, and frequency.

**Prerequisite:**

pip install numpy pandas scikit-learn matplotlib

**Problem Statement**

A dataset that records various parameters affecting the signal quality (Good or Bad). The goal is to develop a k-NN classifier to predict signal quality based on these parameters.

**Program Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler

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

import seaborn as sns

# Example dataset: Distance (meters), Signal Strength (dBm), Frequency (MHz) vs. Signal Quality

data = {

'Distance': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 3, 4, 5, 6],

'Signal\_Strength': [-30, -35, -40, -45, -50, -55, -60, -65, -70, -75, -33, -38, -43, -48, -53],

'Frequency': [850, 850, 850, 850, 850, 1900, 1900, 1900, 1900, 1900, 850, 850, 1900, 1900, 1900],

'Signal\_Quality': ['Good', 'Good', 'Good', 'Good', 'Bad', 'Bad', 'Bad', 'Bad', 'Bad', 'Bad', 'Good', 'Good', 'Bad', 'Bad', 'Bad']

}

# Convert the data into a DataFrame

df = pd.DataFrame(data)

# Separate features and target variable

X = df[['Distance', 'Signal\_Strength', 'Frequency']].values # Features

y = df['Signal\_Quality'].values # Target

# Encode the target variable

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y = le.fit\_transform(y) # 'Good' -> 1, 'Bad' -> 0

# 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)

# Standardize the features

scaler = StandardScaler()

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

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

# Create and train the k-NN classifier

k = 3 # Number of neighbors

model = KNeighborsClassifier(n\_neighbors=k)

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

# Make predictions

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

# Evaluate the model

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

report = classification\_report(y\_test, y\_pred, target\_names=['Bad', 'Good'])

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

print('Classification Report:')

print(report)

# Confusion Matrix

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

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Bad', 'Good'], yticklabels=['Bad', 'Good'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**Output:**



|  |  |
| --- | --- |
| **Exp no: 11** | **Study of Artificial Neural Network (ANN) and Simple Program in ANN** |

**Artificial Neural Networks (ANNs)** are computational models inspired by the human brain's neural networks.



**The given figure illustrates the typical diagram of Biological Neural Network.**

**The typical Artificial Neural Network looks something like the given figure.**



Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

Biological Neural Network Artificial Neural Network

Dendrites Inputs

Cell nucleus Nodes

Synapse Weights

Axon Output

They are used in various applications, from image and speech recognition to game playing and medical diagnosis. Here's a concise guide on the study of ANNs, covering key concepts, components, types, and applications.

**Key Concepts**

**Neuron (Perceptron):** The basic unit of an ANN, analogous to a biological neuron. It takes multiple inputs, applies weights to them, sums them up, and passes the result through an activation function to produce an output.

**Weights:** Parameters that adjust the strength of the connection between neurons. They are crucial in learning and adjusting the network during training.

**Activation Function**: A function applied to the input sum of a neuron to introduce non-linearity. Common functions include sigmoid, tanh, and ReLU (Rectified Linear Unit).

**Layers:** Neurons are organized into layers. The main types are:

Input Layer: Receives initial data.

Hidden Layers: Intermediate layers that process inputs from the input layer. Deep networks have multiple hidden layers.

Output Layer: Produces the final result.

**Forward Propagation**: The process of passing inputs through the network to get the output.

**Backpropagation:** A training method where errors are propagated back through the network to update weights. It involves computing the gradient of the loss function with respect to each weight.

**Loss Function**: A function that measures the difference between the network's output and the actual target values. Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy for classification.

**Types of Neural Networks**

**Feedforward Neural Networks (FNNs):** The simplest type where connections between the nodes do not form a cycle. Information moves in one direction from input to output.

**Convolutional Neural Networks (CNNs):** Specialized for processing data with a grid-like topology, such as images. They use convolutional layers to automatically detect spatial hierarchies in data.

**Recurrent Neural Networks (RNNs):** Designed for sequential data, such as time series or natural language. They have connections that form directed cycles, allowing them to maintain a memory of previous inputs.

**Long Short-Term Memory Networks (LSTMs):** A type of RNN designed to overcome the limitations of traditional RNNs in learning long-term dependencies.

**Generative Adversarial Networks (GANs):** Consist of two networks, a generator and a discriminator, that compete against each other to produce high-quality synthetic data.

**Applications**

**Image and Speech Recognition:** CNNs are extensively used in image recognition, while RNNs and LSTMs are used in speech recognition and natural language processing.

**Medical Diagnosis:** ANNs help in diagnosing diseases by analyzing medical images, genetics, and patient data.

**Autonomous Vehicles**: Used for object detection, lane detection, and decision-making in self-driving cars.

**Financial Services**: Applications include fraud detection, algorithmic trading, and risk management.

**Gaming:** ANNs have been used to develop agents that can play games at a superhuman level.

**Program code:**

import numpy as np

# Define the sigmoid activation function

def sigmoid(x):

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

# Initialize weights with random values

weights = np.random.rand(2, 1) # 2 inputs, 1 output neuron

# Training data (inputs and desired outputs)

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

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

# Learning rate

learning\_rate = 0.1

# Training loop (multiple iterations)

for epoch in range(1000):

# Forward propagation

z = np.dot(inputs, weights) # Dot product of inputs and weights

predictions = sigmoid(z)

# Calculate the error

error = outputs - predictions

# Backpropagation

delta = error \* predictions \* (1 - predictions)

weight\_delta = np.dot(inputs.T, delta)

# Update weights

weights += learning\_rate \* weight\_delta

# Test the network with a new input

new\_input = np.array([1, 0])

prediction = sigmoid(np.dot(new\_input, weights))

print("Predicted output for [1, 0]:", prediction)

**Result:**

Predicted output for [1, 0]: [0.92406673]

|  |  |
| --- | --- |
| **Exp no: 12** | **Study Of Support Vector Machine and and Simple Program in SVM** |

Support Vector Machines (SVMs) are a powerful set of supervised learning algorithms used for classification, regression, and outliers detection. They are particularly well-known for their application in classification problems. Here's a comprehensive guide to understanding SVMs:

**Basics of SVM**

1. Objective: SVM aims to find the best boundary (hyperplane) that separates data points of different classes with the maximum margin. The margin is the distance between the hyperplane and the nearest data point from either class.
2. Hyperplane: In a two-dimensional space, this is a line, but in higher dimensions, it becomes a plane or hyperplane. The goal is to identify the hyperplane that best separates the classes.
3. Support Vectors: These are the data points closest to the hyperplane and are critical in defining the position and orientation of the hyperplane. The model's performance is highly dependent on these support vectors.

**Mathematical Formulation**



**Kernel Trick**

For datasets that are not linearly separable, SVM can be extended to handle this using the kernel trick. Kernels map the original input space into a higher-dimensional space where a linear separation is possible.

Common kernels include:



**SVM for Regression (SVR)**

Support Vector Regression (SVR) uses the same principles as SVM for classification but adapts them for regression. Instead of finding a hyperplane that separates the data, SVR finds a function that deviates from the actual data points by a value no greater than a certain threshold (epsilon).

**Applications of SVM**

1. Text and Hypertext Categorization: Due to their capability to handle high-dimensional spaces efficiently.
2. Image Classification: SVMs are often used for face detection and other image classification tasks.
3. Bioinformatics: For protein classification and gene expression data analysis.
4. Handwriting Recognition: Classifying characters in handwriting recognition systems.

**Advantages and Disadvantages**

Advantages:

* Effective in high-dimensional spaces.
* Still effective when the number of dimensions is greater than the number of samples.
* Memory efficient due to the use of support vectors.
* Versatile with different kernel functions.

Disadvantages:

* Not suitable for large datasets as the training time can be high.
* Less effective on noisy data where classes are not well-separated.
* Choice of kernel and parameters (such as C and γ) requires careful tuning and cross-validation.

**Import Libraries:**

*from sklearn import datasets*

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

*from sklearn.svm import SVC*

*from sklearn.metrics import accuracy\_score*

datasets: This module from Scikit-learn provides various built-in datasets like the Iris dataset.

train\_test\_split: This function is used to split the dataset into training and testing sets.

SVC: This is the SVM classifier class.

accuracy\_score: This function computes the accuracy of the predictions.

**Load Dataset:**

*iris = datasets.load\_iris()*

*X = iris.data*

*y = iris.target*

datasets.load\_iris(): Loads the Iris dataset, which is a classic dataset in machine learning with 150 samples of iris flowers. Each sample has four features (sepal length, sepal width, petal length, petal width) and a corresponding class label (0, 1, or 2) representing three species of iris flowers.

X = iris.data : Assigns the features (input data) to X.

y = iris.target : Assigns the labels (target data) to y.

**Split the Dataset:**

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

train\_test\_split(X, y, test\_size=0.3, random\_state=42): Splits the data into training and testing sets. test\_size=0.3 means 30% of the data will be used for testing, and random\_state=42 ensures reproducibility of the split.

**Create a SVM Classifier**:

*clf = SVC(kernel='linear', C=1)*

SVC(kernel='linear', C=1): Initializes the SVM classifier with a linear kernel and a regularization parameter C set to 1. The kernel='linear' means it will use a linear hyperplane for classification. The parameter C controls the trade-off between achieving a low error on the training data and minimizing the norm of the weights (i.e., the margin).

**Train the Classifier:**

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

*clf.fit(X\_train, y\_train): Trains the SVM classifier using the training data (X\_train and y\_train).*

**Make Predictions:**

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

y\_pred = clf.predict(X\_test): Uses the trained classifier to make predictions on the test data (X\_test). The predictions are stored in y\_pred.

**Evaluate the Model:**

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

*print(f'Accuracy: {accuracy:.2f}')*

accuracy = accuracy\_score(y\_test, y\_pred): Computes the accuracy of the model by comparing the true labels (y\_test) with the predicted labels (y\_pred).

print(f'Accuracy: {accuracy:.2f}'): Prints the accuracy score, formatted to two decimal places.

**Result:**

The experiment successfully demonstrated the application of SVM for classification, showcasing its strengths in handling high-dimensional spaces and providing a clear understanding of its working mechanism.