Pradnya Niketan Education Society's
NAGESH KARAJAGI *ORCHID* COLLEGE OF ENGINEERING & TECHNOLOGY, SOLAPUR
NAAC Accredited, Approved by AICTE, New Delhi & Affiliated to DBATU, Lonere
E-mail : office@orchidengg.ac.in, Website : www.orchidengg.ac.in, Phone No. 9423084363
Post Box No. 154, Gut No. 16, Solapur-Tuljapur Road, Tale Hipparaga, Solapur- 413 002.

Department of Artificial Intelligence & Data Science Engineering

LABORATORY MANUAL

Machine Learning Lab

Semester: V

Subject Code: BTAIL506

Prepared by,

Prof. N. B. Aherwadi

Pradnya Niketan Education Society, Pune.

NAGESH KARAJAGI *ORCHID* COLLEGE OF

ENGINEERING & TECHNOLOGY

SOLAPUR.

INDEX

Lab 1: Python Libraries for Data Science
a.Pandas Library
b. Numpy Library
c. Scikit Learn Library
d. Matplotlib

Aim: Implementation of Data frames using Pandas Library and plots the different charts of data frame using the Matplotlib Library

Theory:

Different Types of Plots

1. Bar Graph: A bar graph is a common type of chart that presents categorical data with rectangular bars. Each bar's length or height corresponds to the value it represents. Bar graphs can be used to compare values across different categories or show the distribution of a single categorical variable.
   Syntax:
   plt.bar(): Creates a bar graph with categories on the x-axis and values on the y-axis.
   color='skyblue': Sets the color of the bars.
   plt.title(): Adds a title to the graph.
   plt.xlabel() and plt.ylabel(): Adds labels to the x and y-axes.

2. Pie Chart: A pie chart is a circular statistical graphic that is divided into slices to illustrate numerical proportions. Each slice represents a proportionate part of the whole data set.
   Syntax:
   plt.pie(): Creates a pie chart with values represented by the slices.
   labels=df['Categories']: Assigns labels to each slice based on the categories in the DataFrame.
   autopct='%1.1f%%': Displays the percentage value on each slice.
   startangle=90: Rotates the pie chart by 90 degrees, starting from the top.
   colors: Sets custom colors for each slice.

3. Box Plot: A box plot, also known as a box-and-whisker plot, is a graphical representation that summarizes the distribution of a dataset. It displays the minimum, first quartile (Q1), median (or second quartile, Q2), third quartile (Q3), and maximum of a set of data. It is particularly useful for identifying the central tendency and spread of the data, as well as detecting potential outliers.
   Syntax:
   plt.boxplot(): Creates a horizontal box plot with values from the DataFrame.
   vert=False: Displays the box plot horizontally.
   plt.title(): Adds a title to the plot.
   plt.xlabel(): Adds a label to the x-axis.
4. Histogram: A histogram is a graphical representation of the distribution of a dataset. It shows the frequency of values within specified intervals, known as bins.
   Syntax:
   plt.hist(): Creates a histogram with values from the DataFrame.
   bins=5: Divides the range of values into 5 bins.
   color='skyblue': Sets the color of the bars.
   edgecolor='black': Sets the color of the edges of the bars.
   plt.title(): Adds a title to the plot.
   plt.xlabel() and plt.ylabel(): Add labels to the x and y-axes.
5. Line Chart and Subplots: Creating a line chart with subplots involves creating multiple plots within the same figure. Subplots are useful when you want to visualize different aspects of the data side by side

Syntax:

plt.subplots(): Creates a figure and a set of subplots. The nrows and ncols parameters define the number of rows and columns of subplots.
fig, axes: Unpacks the returned Figure and Axes objects.
axes[0] and axes[1]: Refer to the individual subplots.
plt.tight_layout(): Adjusts the spacing between subplots to prevent overlap.
plt.show(): Displays the subplots.

Program and Output:

```
import pandas as pd
import matplotlib.pyplot as plt

# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
        'Age': [25, 30, 22, 35, 28],
        'Salary': [50000, 60000, 45000, 70000, 55000]}

df = pd.DataFrame(data)

# Displaying the DataFrame
print("DataFrame:")
print(df)

# Plotting a bar chart for Salary
plt.figure(figsize=(8, 6))
plt.bar(df['Name'], df['Salary'], color='skyblue')
plt.title('Salary Distribution')
plt.xlabel('Name')
plt.ylabel('Salary')
plt.show()

# Plotting a pie chart for Age
plt.figure(figsize=(8, 8))
plt.pie(df['Age'], labels=df['Name'], autopct='%1.1f%%', startangle=90)
plt.title('Age Distribution')
plt.show()
```

Lab 2: Evaluation Metrics
a.Accuracy
b. Precision
c. Recall
d. F1-Score.

Aim: Study of Evaluation Metrics – Accuracy, Precision, Recall, F1-Score.

Theory:

A. Confusion Matrix:

A Confusion matrix is an N x N matrix used for evaluating the performance of a classification model, where N is the number of target classes. The matrix compares the actual target values with those predicted by the machine learning model. A confusion matrix is a tabular summary of the number of correct and incorrect predictions made by a classifier. It is used to measure the performance of a classification model.

The following 4 are the basic terminology used to determine classification metrics
True Positives (TP): when the actual value is Positive and predicted is also Positive.
True negatives (TN): when the actual value is Negative and prediction is also Negative.
False positives (FP): When the actual is negative but prediction is Positive. Also known as the Type 1 error
False negatives (FN): When the actual is Positive but the prediction is Negative. Also known as the Type 2 error

B. Classification Accuracy:

Accuracy simply measures how often the classifier makes the correct prediction. It's the ratio between the number of correct predictions and the total number of predictions. The accuracy metric is not suited for imbalanced classes. Accuracy has its own disadvantages, for imbalanced data, when the model predicts that each point belongs to the majority class label, the accuracy will be high. But, the model is not accurate.
It is a measure of correctness that is achieved in true prediction. In simple words, it tells us how many predictions are actually positive out of the entire total positive predicted.
Accuracy is a valid choice of evaluation for classification problems which are well balanced and not skewed or there is no class imbalance.

C. Classification Report:

a. Precision:

It is a measure of correctness that is achieved in true prediction. In simple words, it tells us how many predictions are actually positive out of the entire total positive predicted.
Precision is defined as the ratio of the total number of correctly classified positive classes divided by the total number of predicted positive classes. Or, out of all the predictive positive classes, how much we predicted correctly. The precision value lies between 0 and 1. Precision should be high (ideally 1).
"Precision is a useful metric in cases where False Positive is a higher concern than False Negatives"

b. Recall or Sensitivity:

It is a measure of actual observations which are predicted correctly, i.e. how many observations of positive class are actually predicted as positive. It is also known as Sensitivity. Recall is a valid choice of evaluation metric when we want to capture as many positives as possible.

Recall is defined as the ratio of the total number of correctly classified positive classes divide by the total number of positive classes. Or, out of all the positive classes, how much we have predicted correctly. Recall should be high (ideally 1).
“Recall is a useful metric in cases where False Negative trumps False Positive”

c. Support:
Support is the number of actual occurrences of the class in the specified dataset. Imbalanced support in the training data may indicate structural weaknesses in the reported scores of the classifier and could indicate the need for stratified sampling or rebalancing. Support doesn’t change between models but instead diagnoses the evaluation process.

d. F1 Score:
The F1 score is a number between 0 and 1 and is the harmonic mean of precision and recall. We use harmonic mean because it is not sensitive to extremely large values, unlike simple averages.
F1 score sort of maintains a balance between the precision and recall for your classifier. If your precision is low, the F1 is low and if the recall is low again your F1 score is low.

Program and Output:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
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)

# Instantiate and train a classifier (e.g., K-Nearest Neighbors)
classifier = KNeighborsClassifier()
classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = classifier.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)
```

Lab 3: Train and Test Sets by Splitting Learn and Test Data..

Aim: Study of Train and Test Sets by Splitting Learn and Test Data..

Theory:

from sklearn.model_selection import train_test_split

Assuming you have features (X) and labels (y)
X, y = ...

Split the data into training and testing sets (e.g., 80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

'test_size' specifies the proportion of the dataset to include in the test split
'random_state' ensures reproducibility by fixing the random seed

Now you can use X_train and y_train for training your model
And X_test and y_test for evaluating its performance

In this example:

X is your feature matrix.
y is your target variable (labels).
train_test_split is a function from scikit-learn that shuffles and splits the dataset into training and testing sets.
Adjust the test_size parameter to control the proportion of data allocated to the testing set. A common split is 80% training and 20% testing, but this can vary depending on your specific needs.

After splitting, you can train your machine learning model on X_train and y_train and evaluate its performance on X_test and y_test. This approach helps ensure that your model is capable of making accurate predictions on new, unseen data

Program and Output:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Display the shapes of the resulting sets
print("Training set shapes:")
print("X_train:", X_train.shape)
print("y_train:", y_train.shape)

print("\nTesting set shapes:")
print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

Lab 4: Linear Regression.

Aim: Study of the Linear Regression Algorithm**.**

Theory:

Simple Regression Calculation
To calculate best-fit line linear regression uses a traditional slope-intercept form which is given below,
$Yi = \beta 0 + \beta 1 Xi$
where Yi = Dependent variable, $\beta 0$ = constant/Intercept, $\beta 1$ = Slope/Intercept, Xi = Independent variable.
This algorithm explains the linear relationship between the dependent(output) variable y and the independent(predictor) variable X using a straight line Y= B0 + B1 X.
Simple Linear Regression explanation

The goal of the linear regression algorithm is to get the best values for B0 and B1 to find the best fit line. The best fit line is a line that has the least error which means the error between predicted values and actual values should be minimum.

Random Error(Residuals)
In regression, the difference between the observed value of the dependent variable(yi) and the predicted value(predicted) is called the residuals.

$\varepsilon i =$ ypredicted – yi

where ypredicted = B0 + B1 Xi

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Load the Boston Housing dataset
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Linear Regression model
model = LinearRegression()

# Train the model on the training set
model.fit(X_train, y_train)

# Make predictions on the testing set
```

```
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Display the evaluation metrics
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-squared (R2): {r2:.4f}")

# Plotting predicted vs actual values
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Linear Regression: Actual vs Predicted Prices")
plt.show()
```

Lab 5: Multivariable Regression.

Aim: To Study Multivariable Regression Algorithm.

Theory, Program and Output:

Multivariable regression, also known as multiple linear regressions, is an extension of simple linear regression that involves modeling the relationship between multiple independent variables (features) and a dependent variable (target). In the context of the Boston Housing dataset, where you have multiple features, here's an example of performing multivariable regression using scikit-learn:

```
import pandas as pd
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the Boston Housing dataset
boston = load_boston()
X = pd.DataFrame(boston.data, columns=boston.feature_names)
y = boston.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Linear Regression model
model = LinearRegression()

# Train the model on the training set
model.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Display the evaluation metrics
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-squared (R2): {r2:.4f}")

# Display the coefficients for each feature
coefficients = pd.DataFrame({'Feature': X.columns, 'Coefficient': model.coef_})
print("\nCoefficients:")
print(coefficients)
```

Lab 6: Decision Tree Algorithm implementation.

Aim: Study of Decision Tree Algorithm implementation.

Theory:

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes and leaf nodes.

Decision tree learning employs a divide and conquer strategy by conducting a greedy search to identify the optimal split points within a tree. This process of splitting is then repeated in a top-down, recursive manner until all, or the majority of records have been classified under specific class labels. Whether or not all data points are classified as homogenous sets is largely dependent on the complexity of the decision tree. Smaller trees are more easily able to attain pure leaf nodes

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn import tree
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
```

```
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Decision Tree classifier
classifier = DecisionTreeClassifier()

# Train the classifier on the training set
classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)

# Visualize the decision tree (optional)
plt.figure(figsize=(12, 8))
tree.plot_tree(classifier, feature_names=iris.feature_names, class_names=iris.target_names,
filled=True)
plt.show()
```

- The Iris dataset is loaded, and the features are stored in X, and target values are stored in y.
- The data is split into training and testing sets using train_test_split.
- A Decision Tree classifier is instantiated and trained on the training set.
- Predictions are made on the testing set, and evaluation metrics (accuracy, confusion matrix, classification report) are calculated.
- Optionally, you can visualize the decision tree using Matplotlib

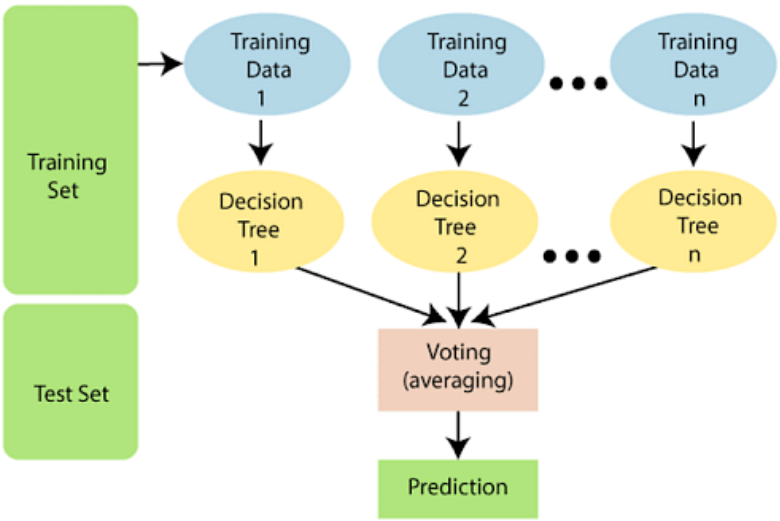
Lab 7: Random Forest Algorithm implementation

Aim: To Study Random Forest Algorithm implementation.

Theory:

A Random Forest Algorithm is a supervised machine learning algorithm that is extremely popular and is used for Classification and Regression problems in Machine Learning. We know that a forest comprises numerous trees, and the more trees more it will be robust. Similarly, the greater the number of trees in a Random Forest Algorithm, the higher its accuracy and problem-solving ability. Random Forest is a classifier that contains several decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. It is based on the concept of ensemble learning which is a process of combining multiple classifiers to solve a complex problem and improve the performance of the model.

Working of Random Forest Algorithm



Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Random Forest classifier
classifier = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
# Train the classifier on the training set
classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)
```

Lab 8: Naive Bayes Classification Algorithm implementation.

Aim: To Study Naive Bayes Classification Algorithm implementation.

Theory:

The Naïve Bayes classifier is a supervised machine learning algorithm, which is used for classification tasks, like text classification. It is also part of a family of generative learning algorithms, meaning that it seeks to model the distribution of inputs of a given class or category. Unlike discriminative classifiers, like logistic regression, it does not learn which features are most important to differentiate between classes.

$$P(Y|X) = \frac{P(X \text{ and } Y)}{P(X)}$$

Bayes' Theorem is distinguished by its use of sequential events, where additional information later acquired impacts the initial probability. These probabilities are denoted as the prior probability and the posterior probability. The prior probability is the initial probability of an event before it is contextualized under a certain condition, or the marginal probability. The posterior probability is the probability of an event after observing a piece of data.

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Naive Bayes classifier (Gaussian Naive Bayes for continuous features)
classifier = GaussianNB()

# Train the classifier on the training set
classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

```
# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)
```

- The Iris dataset is loaded, and the features are stored in X, and target values are stored in y.
- The data is split into training and testing sets using train_test_split.
- A Gaussian Naive Bayes classifier is instantiated and trained on the training set. The GaussianNB is suitable for datasets with continuous features.
- Predictions are made on the testing set, and evaluation metrics (accuracy, confusion matrix, classification report) are calculated.
- Naive Bayes is a probabilistic algorithm that is often used for classification tasks. The Gaussian Naive Bayes assumes that the features follow a Gaussian distribution. Customize the implementation based on your specific needs and the characteristics of your dataset.

Lab 9: K-Nearest Neighbor Algorithm implementation

Aim: To Study K-Nearest Neighbor Algorithm implementation.

Theory:

This algorithm is used to solve the classification model problems. K-nearest neighbor or K-NN algorithm basically creates an imaginary boundary to classify the data. When new data points come in, the algorithm will try to predict that to the nearest of the boundary line.

Therefore, larger k value means smother curves of separation resulting in less complex models. Whereas, smaller k value tends to overfit the data and resulting in complex models.

Note: It's very important to have the right k-value when analyzing the dataset to avoid overfitting and underfitting of the dataset.

Using the k-nearest neighbor algorithm we fit the historical data (or train the model) and predict the future.

Following steps are performed in Algorithm:

1. The k-nearest neighbor algorithm is imported from the scikit-learn package.
2. Create feature and target variables.
3. Split data into training and test data.
4. Generate a k-NN model using neighbors value.
5. Train or fit the data into the model.
6. Predict the future.

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the K-Nearest Neighbors classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)  #You can adjust the number of neighbors (k)

# Train the classifier on the training set
knn_classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = knn_classifier.predict(X_test)
```

```
# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)
```

- The Iris dataset is loaded, and the features are stored in X, and target values are stored in y.
- The data is split into training and testing sets using train_test_split.
- A K-Nearest Neighbors classifier is instantiated with the number of neighbors set to 3 (you can adjust n_neighbors based on your needs).
- The classifier is trained on the training set.
- Predictions are made on the testing set, and evaluation metrics (accuracy, confusion matrix, classification report) are calculated.
- K-Nearest Neighbors is a simple and effective algorithm for classification tasks. It classifies a data point based on the majority class among its k-nearest neighbors. Adjust the parameters and customize the implementation based on your specific needs and the characteristics of your dataset.

Lab 10: SVM Algorithm implementation.

Aim: To Study SVM Algorithm implementation.

Theory:

Support Vector Machine (SVM) is a powerful machine learning algorithm used for linear or nonlinear classification, regression, and even outlier detection tasks. SVMs can be used for a variety of tasks, such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection. SVMs are adaptable and efficient in a variety of applications because they can manage high-dimensional data and nonlinear relationships.
SVM algorithms are very effective as we try to find the maximum separating hyperplane between the different classes available in the target feature.

Support Vector Machine Terminology:

1. Hyperplane: Hyperplane is the decision boundary that is used to separate the data points of different classes in a feature space. In the case of linear classifications, it will be a linear equation i.e. wx+b = 0.
2. Support Vectors: Support vectors are the closest data points to the hyperplane, which makes a critical role in deciding the hyperplane and margin.
3. Margin: Margin is the distance between the support vector and hyperplane. The main objective of the support vector machine algorithm is to maximize the margin. The wider margin indicates better classification performance.
4. Kernel: Kernel is the mathematical function, which is used in SVM to map the original input data points into high-dimensional feature spaces, so, that the hyperplane can be easily found out even if the data points are not linearly separable in the original input space. Some of the common kernel functions are linear, polynomial, radial basis function(RBF), and sigmoid.
5. Hard Margin: The maximum-margin hyperplane or the hard margin hyperplane is a hyperplane that properly separates the data points of different categories without any misclassifications.
6. Soft Margin: When the data is not perfectly separable or contains outliers, SVM permits a soft margin technique. Each data point has a slack variable introduced by the soft-margin SVM formulation, which softens the strict margin requirement and permits certain misclassifications or violations. It discovers a compromise between increasing the margin and reducing violations.
7. C: Margin maximisation and misclassification fines are balanced by the regularisation parameter C in SVM. The penalty for going over the margin or misclassifying data items is decided by it. A stricter penalty is imposed with a greater value of C, which results in a smaller margin and perhaps fewer misclassifications.
8. Hinge Loss: A typical loss function in SVMs is hinge loss. It punishes incorrect classifications or margin violations. The objective function in SVM is frequently formed by combining it with the regularisation term.
9. Dual Problem: A dual Problem of the optimisation problem that requires locating the Lagrange multipliers related to the support vectors can be used to solve SVM. The dual formulation enables the use of kernel tricks and more effective computing.

Program and Output:

```
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 the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Instantiate the Support Vector Machine classifier
svm_classifier = SVC(kernel='linear')  # You can choose different kernels like 'linear', 'poly', 'rbf', etc.

# Train the classifier on the training set
svm_classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = svm_classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display the results
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", classification_rep)
```