REGRESSION

```
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 sample data
X, y = make_regression(n_samples=100, n_features=1, noise=10, 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)

# Initialize and fit the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Calculate mean squared error
mse_train = mean_squared_error(y_train, y_pred_train)
mse_test = mean_squared_error(y_test, y_pred_test)

# Plot the data and the regression line
plt.scatter(X_train, y_train, color='blue', label='Training data')
plt.scatter(X_test, y_test, color='red', label='Testing data')
plt.plot(X_train, y_pred_train, color='green', label='Regression line')
plt.title('Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

# Print mean squared error
print("Mean Squared Error (Training):", mse_train)
print("Mean Squared Error (Testing):", mse_test)
```

PCA

```
from sklearn.datasets import load_iris

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt


# Load the Iris dataset

iris = load_iris()

X = iris.data

y = iris.target


# Perform PCA

pca = PCA(n_components=2)  # Reduce to 2 principal components

X_pca = pca.fit_transform(X)


# Plot the PCA-transformed data

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

for i in range(3):

    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], label=iris.target_names[i])

plt.title('PCA of Iris Dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.legend()

plt.show()
```

SVM –

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn import svm

# Generate synthetic dataset with blobs
X, y = make_blobs(n_samples=100, centers=2, random_state=42)

# Print the dataset
print("Dataset:")
print("Number of samples: ", len(X))

# Train SVM model
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

# Plot data points
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, s=30, edgecolors='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Plot decision boundary
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T  # Mesh grid for the first two features
Z = clf.decision_function(xy).reshape(XX.shape)

# Plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
        linestyles=['--', '-', '--'])
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
        linewidth=1, facecolors='none', edgecolors='k')
plt.title('SVM Decision Boundary')
plt.show()
```

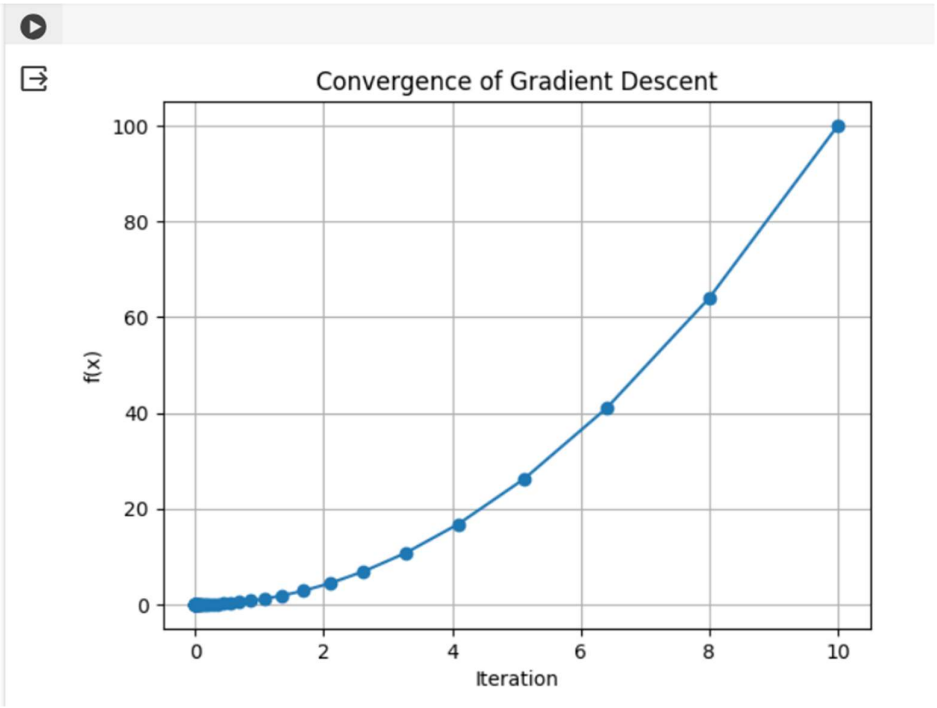
LAB 5: IMPLEMENTATION OF GRADIENT DESCENT ALGORITHM

Gradient descent: An optimization algorithm that iteratively adjusts parameters to minimize a given function by moving in the direction of the steepest descent. It's a core method in machine learning for training models and finding optimal solutions.

Code-

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return x**2

# Define the gradient of the function
def grad_f(x):
    return 2*x

# Gradient Descent Algorithm
def gradient_descent(learning_rate, iterations):
    x = 10  # Initial guess
    history = [x]
    for i in range(iterations):
        gradient = grad_f(x)
        x = x - learning_rate * gradient
        history.append(x)
    return history

# Setting the hyperparameters
learning_rate = 0.1
iterations = 50

# Running the gradient descent
history = gradient_descent(learning_rate, iterations)

# Visualizing the convergence
plt.plot(history, f(np.array(history)), '-o')
plt.xlabel('Iteration')
plt.ylabel('f(x)')
plt.title('Convergence of Gradient Descent')
plt.grid(True)
plt.show()
```

Output-


Convergence of Gradient Descent
f(x)
Iteration
0
20
40
60
80
100
0
2
4
6
8
10

LAB 6: IMPLEMENTATION OF NAÏVE BAYESIAN CLASSIFICATION

Naive Bayesian classification is a probabilistic machine learning technique based on Bayes' theorem, assuming independence among features, commonly used for classification tasks where each feature contributes independently to the prediction of the class label.

Code-

```
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
import numpy as np

iris = load_iris()
X = iris.data
y = iris.target

# Splitting the dataset 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)

# Initializing Gaussian Naive Bayes classifier
clf = GaussianNB()

# Training the classifier
clf.fit(X_train, y_train)

# Calculating class priors
class_counts = np.bincount(y_train)
class_priors = class_counts / len(y_train)

# Making predictions using Bayesian Decision Theory
posterior_probs = clf.predict_proba(X_test)
y_pred = np.argmax(posterior_probs * class_priors, axis=1)
print("Predicted Labels:", y_pred)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output-

```
Predicted Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
Accuracy: 1.0
```

LAB 7: IMPLEMENTATION OF DECISION TREE CLASSIFIER

A decision tree classifier is a supervised machine learning algorithm that recursively partitions the feature space into regions, forming a tree-like structure, where each internal node represents a decision based on a feature, and each leaf node represents a class label.

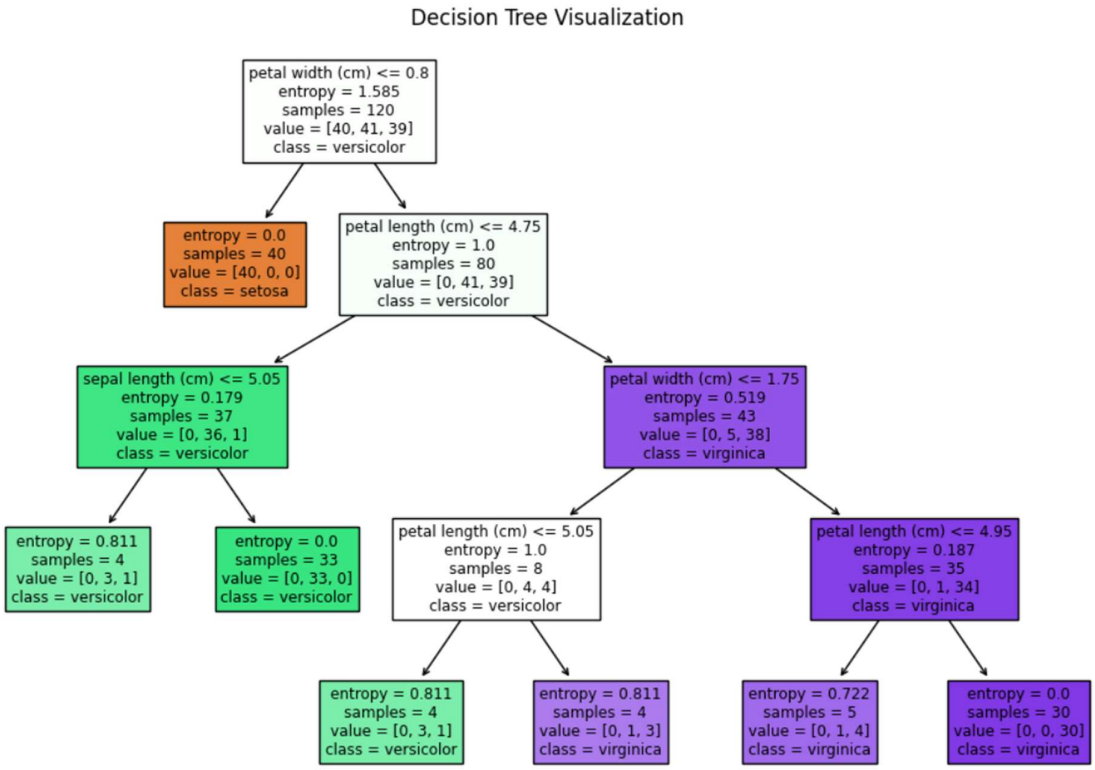
Code-

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data
y = iris.target

# Splitting the dataset 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)

# Initializing Decision Tree Classifier
dt_classifier = DecisionTreeClassifier()

# Defining hyperparameters grid for tuning
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Performing grid search cross-validation to find the best hyperparameters
grid_search = GridSearchCV(dt_classifier, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Getting the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Initializing Decision Tree Classifier with best hyperparameters
best_dt_classifier = DecisionTreeClassifier(**best_params)

# Training the classifier with the best hyperparameters
best_dt_classifier.fit(X_train, y_train)

# Visualizing the constructed decision tree
```

```
plt.figure(figsize=(12, 8))
plot_tree(best_dt_classifier, filled=True, feature_names=iris.feature_names,
    class_names=iris.target_names)
plt.title("Decision Tree Visualization")
plt.show()
```

Output-

```
Best Hyperparameters: {'criterion': 'entropy', 'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 2}
```



LAB 8: IMPLEMENTATION OF BASIC ENSEMBLE METHOD AND BOOSTING METHOD

Ensemble methods combine multiple models to improve prediction accuracy, such as Random Forest, which aggregates the predictions of multiple decision trees. Boosting is a sequential ensemble technique where each subsequent model focuses on correcting the errors made by the previous ones, as seen in AdaBoost and XGBoost, leading to improved overall performance.

Code-

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
X = iris.data
y = iris.target

# Splitting the dataset 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)

# Initializing individual models
random_forest = RandomForestClassifier(random_state=42)
adaboost = AdaBoostClassifier(random_state=42)
xgboost = XGBClassifier(random_state=42)

# Training individual models
random_forest.fit(X_train, y_train)
adaboost.fit(X_train, y_train)
xgboost.fit(X_train, y_train)

# Making predictions
rf_predictions = random_forest.predict(X_test)
adaboost_predictions = adaboost.predict(X_test)
xgboost_predictions = xgboost.predict(X_test)

# Calculating accuracies
rf_accuracy = accuracy_score(y_test, rf_predictions)
adaboost_accuracy = accuracy_score(y_test, adaboost_predictions)
xgboost_accuracy = accuracy_score(y_test, xgboost_predictions)

print("Random Forest Accuracy:", rf_accuracy)
print("AdaBoost Accuracy:", adaboost_accuracy)
print("XGBoost Accuracy:", xgboost_accuracy)
```

Output-

```
Random Forest Accuracy: 1.0
AdaBoost Accuracy: 1.0
XGBoost Accuracy: 1.0
```

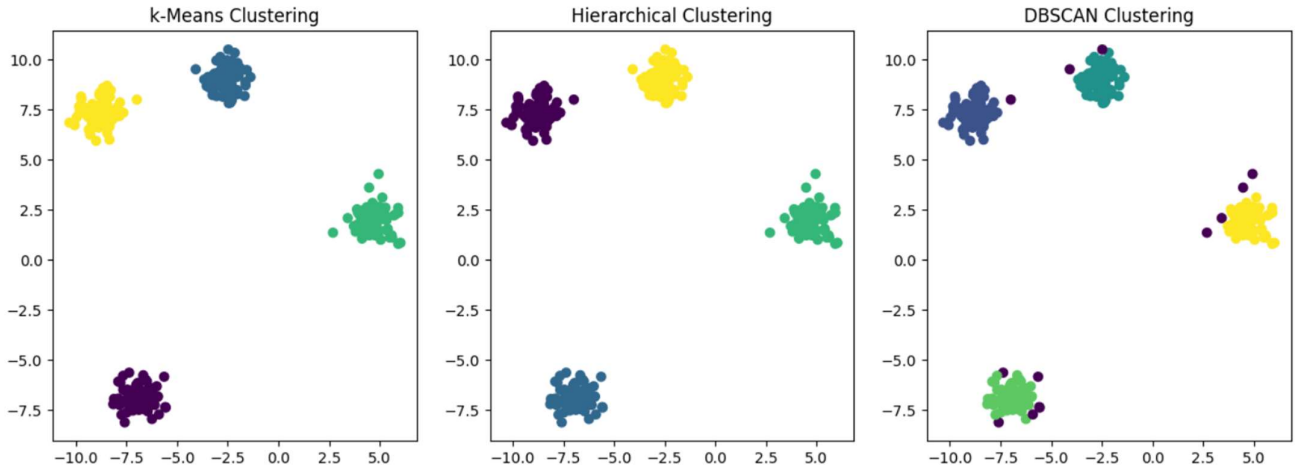
LAB 9: IMPLEMENTATION OF CLUSTERING METHODS

Clustering algorithms such as KMeans, Hierarchical, and DBSCAN are fundamental techniques in unsupervised machine learning, used to partition datasets into distinct groups based on similarity measures, with KMeans clustering assigning data points to clusters by minimizing the within-cluster variance, Hierarchical clustering forming nested clusters by iteratively merging or splitting clusters, and DBSCAN identifying dense regions of data points, effectively handling noise and arbitrary cluster shapes.

Code-

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN

# Generating synthetic data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=42)

# Implementing k-Means clustering
kmeans = KMeans(n_clusters=4)
kmeans_labels = kmeans.fit_predict(X)

# Implementing Hierarchical clustering (Agglomerative)
hierarchical = AgglomerativeClustering(n_clusters=4)
hierarchical_labels = hierarchical.fit_predict(X)

# Implementing DBSCAN clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_labels = dbscan.fit_predict(X)

# Visualizing clustering results
plt.figure(figsize=(15, 5))

plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis')
plt.title("k-Means Clustering")

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=hierarchical_labels, cmap='viridis')
plt.title("Hierarchical Clustering")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=dbscan_labels, cmap='viridis')
plt.title("DBSCAN Clustering")

plt.show()
```

Output-


k-Means Clustering
Hierarchical Clustering
DBSCAN Clustering
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0

LAB 10: IMPLEMENTATION OF K-NEAREST NEIGHBORS (KNN)

K-Nearest Neighbors (KNN) is a simple yet powerful non-parametric classification algorithm that predicts the class of a data point based on the majority class of its nearest neighbors in the feature space, making it effective for both classification and regression tasks.

Code-

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Generate synthetic data
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)

# Split the dataset 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)

# Implement KNN classifier
def knn_classifier(k):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    return accuracy_score(y_test, y_pred)

# Implement Decision Tree classifier
def decision_tree_classifier():
    dt = DecisionTreeClassifier(random_state=42)
    dt.fit(X_train, y_train)
    y_pred = dt.predict(X_test)
    return accuracy_score(y_test, y_pred)

# Evaluate KNN classifier with different values of K
k_values = [1, 3, 5, 7, 9]
knn_accuracies = []
for k in k_values:
    accuracy = knn_classifier(k)
    knn_accuracies.append(accuracy)

# Evaluate Decision Tree classifier
dt_accuracy = decision_tree_classifier()

# Print accuracies
print("Decision Tree Accuracy:", dt_accuracy)
print("KNN Accuracies for Different Values of K:")
```

```
for k, accuracy in zip(k_values, knn_accuracies):
    print("K =", k, "Accuracy:", accuracy)
```

Output-

```
Decision Tree Accuracy: 0.875
KNN Accuracies for Different Values of K:
K = 1 Accuracy: 0.78
K = 3 Accuracy: 0.81
K = 5 Accuracy: 0.81
K = 7 Accuracy: 0.81
K = 9 Accuracy: 0.8
```

LAB 11: IMPLEMENTATION OF LOGISTIC REGRESSION

Logistic regression is a statistical model used for binary classification, which predicts the probability of an instance belonging to a particular class based on input features, utilizing the logistic (sigmoid) function to map input values to probabilities.

Code-

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

iris = load_iris()

# Extracting features (X) and target variable (y)
X = iris.data
y = iris.target

# Splitting the dataset 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)

# Standardizing the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initializing and fitting the logistic regression model
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train, y_train)

# Printing some values
print("Coefficients:", log_reg.coef_)
print("Intercept:", log_reg.intercept_)

# Predicting on the test set
y_pred = log_reg.predict(X_test)

# Printing predicted values
print("Predicted values:", y_pred)

# Calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output-

```
Coefficients: [[-1.00316768  1.14456076 -1.81255767 -1.69176083]
 [ 0.52785456 -0.28289055 -0.34085076 -0.71984718]
 [ 0.47531311 -0.8616702   2.15340842  2.41160801]]
Intercept: [-0.13379691  1.98339163 -1.84959472]
Predicted values: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
Accuracy: 1.0
```