**#Principal Component Analysis**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

target\_names = iris.target\_names

# Standardize the features

scaler = StandardScaler()

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

# Perform PCA

pca = PCA(n\_components=2)

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

# Create a DataFrame for the PCA data

pca\_df = pd.DataFrame(data=X\_pca, columns=['PC1', 'PC2'])

pca\_df['Target'] = y

# Plotting the PCA results

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

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

lw = 2

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

plt.scatter(X\_pca[y == i, 0], X\_pca[y == i, 1], color=color, alpha=.8, lw=lw,

label=target\_name)

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

plt.title('PCA of IRIS dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.show()

**Support Vector Machines** (SVM) are powerful supervised learning models used for classification and regression tasks. Here's a brief overview:

* Objective: SVM aims to find the hyperplane that best separates data points belonging to different classes in the feature space. This hyperplane is chosen such that the margin, i.e., the distance between the hyperplane and the nearest data points (support vectors) from each class, is maximized.
* Support Vectors: Support vectors are the data points that lie closest to the decision boundary (hyperplane). They are crucial for defining the decision boundary and optimizing the margin.
* Kernel Trick: SVM can efficiently handle nonlinear decision boundaries by using a kernel function, which implicitly maps the input data into a higher-dimensional space where a linear separation is possible. Common kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid kernels.
* Regularization Parameter (C): SVM has a regularization parameter (C) that controls the trade-off between maximizing the margin and minimizing the classification error. A smaller C value allows for a wider margin but may lead to misclassification, while a larger C value prioritizes correct classification at the expense of a narrower margin.
* Advantages:
* Effective in high-dimensional spaces.
* Versatile with different kernel functions for handling nonlinear data. o Robust against overfitting, especially when using proper regularization.
* Disadvantages:
* Computationally intensive, especially for large datasets.
* Sensitivity to the choice of kernel and hyperparameters.
* Can be challenging to interpret the model's decision boundary, especially in high-dimensional spaces.
* Applications:
* Text classification, image classification, and handwriting recognition. o Bioinformatics (e.g., protein structure prediction).
* Financial forecasting.
* Medical diagnosis.

Overall, SVM is a versatile and effective algorithm for both classification and regression tasks, especially when dealing with high-dimensional or nonlinear data. However, careful parameter tuning and understanding of the data are crucial for achieving optimal performance.

Code-

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import datasets

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

iris = datasets.load\_iris()

X = iris.data[:, :2]

y = iris.target

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

svm = SVC(kernel='linear')

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

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

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

print("Accuracy:", accuracy)

plt.figure(figsize=(10, 6))

for i in range(len(iris.target\_names)):

plt.scatter(X[y == i][:, 0], X[y == i][:, 1], label=iris.target\_names[i])

plt.xlabel('Sepal Length (cm)')

plt.ylabel('Sepal Width (cm)')

plt.title('Sepal Width vs Sepal Length')

plt.legend()

plt.grid(True)

plt.show()

NAÏVE BAYES

**•** Bayes' Theorem: Naive Bayes is based on Bayes' theorem, which is a fundamental theorem in probability theory that describes the probability of an event, based on prior knowledge of conditions related to the event.

**•** Assumption of Feature Independence: Naive Bayes assumes that the presence of a particular feature in a class is independent of the presence of any other feature. This is a strong assumption, and it's often violated in real-world datasets. Despite this, Naive Bayes can still perform well in practice.

**•** Classification: Given a set of features (attributes), Naive Bayes calculates the probability of each class based on these features. It assigns the class with the highest probability as the predicted class for the input data.

**•** Parameter Estimation: In training, Naive Bayes estimates the probability distributions of features for each class from the training data. For example, it calculates the likelihood of a word occurring in a particular class in text classification.

**•** Prediction: In prediction, Naive Bayes uses the estimated probabilities to predict the most probable class for new instances based on the observed features.

Despite its simplicity and the strong independence assumption, Naive Bayes classifiers often perform well in practice, especially in text classification tasks where the feature space is high-dimensional. However, they may not perform well if the independence assumption is

severely violated or if there's a lack of sufficient training data.

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

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

clf = GaussianNB()

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

class\_counts = np.bincount(y\_train)

class\_priors = class\_counts / len(y\_train)

posterior\_probs = clf.predict\_proba(X\_test)

y\_pred = np.argmax(posterior\_probs \* class\_priors, axis=1)

print("Predicted Labels:", y\_pred)

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

print("Accuracy:", accuracy)

**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

def f(x):

return x\*\*2

def grad\_f(x):

return 2\*x

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

learning\_rate = 0.1

iterations = 50

history = gradient\_descent(learning\_rate, iterations)

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()

**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

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

dt\_classifier = DecisionTreeClassifier()

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]

}

grid\_search = GridSearchCV(dt\_classifier, param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_

print("Best Hyperparameters:", best\_params)

best\_dt\_classifier = DecisionTreeClassifier(\*\*best\_params)

best\_dt\_classifier.fit(X\_train, y\_train)

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()

**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

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

random\_forest = RandomForestClassifier(random\_state=42)

adaboost = AdaBoostClassifier(random\_state=42)

xgboost = XGBClassifier(random\_state=42)

random\_forest.fit(X\_train, y\_train)

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

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

rf\_predictions = random\_forest.predict(X\_test)

adaboost\_predictions = adaboost.predict(X\_test)

xgboost\_predictions = xgboost.predict(X\_test)

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)

**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

X, \_ = make\_blobs(n\_samples=300, centers=4, cluster\_std=0.60, random\_state=42)

kmeans = KMeans(n\_clusters=4)

kmeans\_labels = kmeans.fit\_predict(X)

hierarchical = AgglomerativeClustering(n\_clusters=4)

hierarchical\_labels = hierarchical.fit\_predict(X)

dbscan = DBSCAN(eps=0.5, min\_samples=5)

dbscan\_labels = dbscan.fit\_predict(X)

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()

**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

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_classes=2, random\_state=42)

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

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)

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)

k\_values = [1, 3, 5, 7, 9]

knn\_accuracies = []

for k in k\_values:

accuracy = knn\_classifier(k)

knn\_accuracies.append(accuracy)

dt\_accuracy = decision\_tree\_classifier()

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)

**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()

X = iris.data

y = iris.target

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

scaler = StandardScaler()

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

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

log\_reg = LogisticRegression(max\_iter=1000)

log\_reg.fit(X\_train, y\_train)

print("Coefficients:", log\_reg.coef\_)

print("Intercept:", log\_reg.intercept\_)

y\_pred = log\_reg.predict(X\_test)

print("Predicted values:", y\_pred)

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

print("Accuracy:", accuracy)