**Importing datasets**

**Pandas**:

import pandas as pd

data = pd.read\_csv('data.csv')

print(breast\_cancer\_data**.**DESCR) #g

**SKlearn**

from sklearn import datasets

import numpy as np

iris = datasets.load\_iris()

X = iris.data[:, [2, 3]]

y = iris.target

**np.unique(y) function returned the three unique class labels stored in iris.target**

**EDA**

# Import necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

**# Load your dataset (replace 'your\_data.csv' with your actual file path or URL)**

data = pd.read\_csv('your\_data.csv')

**# Display basic information about the dataset**

print("Data Overview:")

print(data.head())

print("\nData Info:")

print(data.info())

print("\nDescriptive Statistics:")

print(data.describe())

print(data.dtypes)

**# Check for missing values**

print("\nMissing Values:")

print(data.isnull().sum())

**# Visualize the distribution of numeric features**

numeric\_features = data.select\_dtypes(include=[np.number])

numeric\_features.hist(bins=20, figsize=(12, 8))

plt.suptitle("Numeric Feature Distributions", y=1.02)

plt.show()

**# Visualize the correlation matrix using a heatmap**

correlation\_matrix = data.corr()

plt.figure(figsize=(12, 8))

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', fmt=".2f")

plt.title("Correlation Matrix")

plt.show()

**# Box plots to identify outliers in numeric features**

plt.figure(figsize=(12, 8))

sns.boxplot(data=numeric\_features)

plt.title("Boxplot of Numeric Features")

plt.show()

**# Pairplot for visualizing relationships between pairs of numeric features**

sns.pairplot(data=numeric\_features)

plt.suptitle("Pairplot of Numeric Features", y=1.02)

plt.show()

**# Countplot for categorical features**

categorical\_features = data.select\_dtypes(include=[np.object])

for column in categorical\_features.columns:

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

sns.countplot(x=column, data=data, palette="viridis")

plt.title(f"Countplot of {column}")

plt.show()

**Visualisations**

**Barplots**

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'column\_name' with the actual column name you want to visualize

bar\_data = df['column\_name'].value\_counts()

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

bar\_data.plot(kind='bar', color='skyblue')

plt.title('Bar Graph')

plt.xlabel('Categories')

plt.ylabel('Counts')

plt.show()

**Histogram**

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'column\_name' with the actual column name you want to visualize

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

plt.hist(df['column\_name'], bins=30, color='green', alpha=0.7)

plt.title('Histogram')

plt.xlabel('Values')

plt.ylabel('Frequency')

plt.show()

Line Graph:

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'x\_column' and 'y\_column' with the actual column names you want to visualize

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

plt.plot(df['x\_column'], df['y\_column'], marker='o', color='orange', linestyle='-', linewidth=2)

plt.title('Line Graph')

plt.xlabel('X-axis Label')

plt.ylabel('Y-axis Label')

plt.show()

Scatter plot:

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'x\_column' and 'y\_column' with the actual column names you want to visualize

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

plt.scatter(df['x\_column'], df['y\_column'], color='red', alpha=0.5)

plt.title('Scatter Plot')

plt.xlabel('X-axis Label')

plt.ylabel('Y-axis Label')

plt.show()

Pairplot

import seaborn as sns

# Assuming 'df' is your DataFrame

# Adjust the list of columns based on what you want to visualize

columns\_to\_plot = ['column1', 'column2', 'column3']

sns.pairplot(df[columns\_to\_plot])

plt.show()

Stacked barplot

import pandas as pd

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'category\_column' and 'value\_columns' with the actual column names you want to visualize

category\_column = 'category\_column'

value\_columns = ['value\_column1', 'value\_column2']

stacked\_data = df[value\_columns].transpose()

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

stacked\_data.plot(kind='bar', stacked=True, color=['skyblue', 'orange'])

plt.title('Stacked Barplot')

plt.xlabel('Categories')

plt.ylabel('Values')

plt.legend(title=category\_column)

plt.show()

Histogram

import seaborn as sns

import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame

# Replace 'column\_name' with the actual column name you want to visualize

column\_name = 'column\_name'

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

sns.histplot(df[column\_name], kde=True, color='skyblue')

plt.title('Histogram with Distribution Curve')

plt.xlabel('Values')

plt.ylabel('Frequency')

plt.show()

**Pre processing**

**Missing Values**

df.isnull().sum()

df.dropna(axis=0) #remove rows w 0

df.dropna(axis=1) #remove columns w 0

from sklearn.impute import SimpleImputer

import numpy as np

# Create a SimpleImputer instance with strategy='mean' or ‘median’ or ‘most frequent’

imputer = SimpleImputer(strategy='mean')

imputed\_data = imputer.fit\_transform(data)

print("\nImputed Data:")

print(imputed\_data)

**Encoding Values**

df\_encoded = pd.get\_dummies(df, columns=['Category'], prefix='Category') #one hot

print(df\_encoded)

from sklearn.preprocessing import LabelEncoder #Label encoder

df = pd.DataFrame(data)

label\_encoder = LabelEncoder()

df['Category\_encoded'] = label\_encoder.fit\_transform(df['Category'])

print(df)

**Scaling and Normalisation**

import pandas as pd

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

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

# Sample DataFrame

data = {'Feature1': [10, 20, 30, 40],

'Feature2': [0.1, 0.2, 0.3, 0.4],

'Target': [1, 0, 1, 0]}

df = pd.DataFrame(data)

# Separate features and target

X = df.drop('Target', axis=1)

y = df['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)

# Define the columns to be scaled and normalized

numeric\_features = X.columns

# Create a ColumnTransformer to apply transformations to specific columns

preprocessor = ColumnTransformer(

transformers=[

('scale', StandardScaler(), numeric\_features), # StandardScaler for scaling

('normalize', MinMaxScaler(), numeric\_features) # MinMaxScaler for normalization

])

# Create a pipeline with the preprocessor and any additional steps (e.g., a classifier)

pipeline = Pipeline([

('preprocessor', preprocessor),

# Add your classifier or other steps here

])

# Fit the pipeline on the training data and transform it

X\_train\_transformed = pipeline.fit\_transform(X\_train)

# Transform the test data using the same pipeline

X\_test\_transformed = pipeline.transform(X\_test)

# Display the transformed data

print("Transformed Training Data:")

print(X\_train\_transformed)

print("\nTransformed Test Data:")

print(X\_test\_transformed)

**Splitting into train test**

import pandas as pd

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

# Assume 'df' is your DataFrame

# Specify the features (X) and the target variable (y)

# Adjust column names accordingly based on your DataFrame structure

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# Split the data into training and testing sets (e.g., 80% train, 20% test)

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 shape (X\_train, y\_train):", X\_train.shape, y\_train.shape)

print("Testing set shape (X\_test, y\_test):", X\_test.shape, y\_test.shape)

* Test feature importance using decision trees in book ch 4

**PCA**

>>> from sklearn.linear\_model import LogisticRegression

>>> from sklearn.decomposition import PCA

>>> pca = PCA(n\_components=2)

>>> lr = LogisticRegression()

>>> X\_train\_pca = pca.fit\_transform(X\_train\_std)

>>> X\_test\_pca = pca.transform(X\_test\_std)

>>> lr.fit(X\_train\_pca, y\_train)

>>> plot\_decision\_regions(X\_train\_pca, y\_train, classifier=lr)

>>> plt.xlabel('PC 1')

>>> plt.ylabel('PC 2')

>>> plt.legend(loc='lower left')

>>> plt.show()

**LDA via scikit-learn**

>>> from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA

>>> lda = LDA(n\_components=2)

>>> X\_train\_lda = lda.fit\_transform(X\_train\_std, y\_train)

>>> lr = LogisticRegression()

>>> lr = lr.fit(X\_train\_lda, y\_train)

>>> plot\_decision\_regions(X\_train\_lda, y\_train, classifier=lr)

>>> plt.xlabel('LD 1') >>> plt.ylabel('LD 2')

>>> plt.legend(loc='lower left')

>>> plt.show()

**Kernel principal component analysis**

>>> from sklearn.decomposition import KernelPCA

>>> X, y = make\_moons(n\_samples=100, random\_state=123)

>>> scikit\_kpca = KernelPCA(n\_components=2, kernel='rbf', gamma=15)

>>> X\_skernpca = scikit\_kpca.fit\_transform(X)

>>> plt.scatter(X\_skernpca[y==0, 0], X\_skernpca[y==0, 1], color='red', marker='^', alpha=0.5)

>>> plt.scatter(X\_skernpca[y==1, 0], X\_skernpca[y==1, 1], color='blue', marker='o', alpha=0.5)

>>> plt.xlabel('PC1')

>>> plt.ylabel('PC2')

>>> plt.show()

**Grid Search CV**

from sklearn.model\_selection import GridSearchCV

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

mv\_clf = Pipeline([

('scaler', StandardScaler()), # Add a scaler if needed

('clf', BaggingClassifier(base\_estimator=DecisionTreeClassifier(), n\_estimators=10)),

])

params = {

'clf\_\_base\_estimator\_\_max\_depth': [1, 2], # Example: For DecisionTreeClassifier

'clf\_\_n\_estimators': [50, 100], # Example: For BaggingClassifier

'scaler\_\_with\_mean': [True, False], # Example: For StandardScaler

}

grid = GridSearchCV(estimator=mv\_clf, param\_grid=params, cv=5, scoring='accuracy')

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

print("Best Parameters:", grid.best\_params\_)

print("Best Accuracy Score:", grid.best\_score\_)

best\_model = grid.best\_estimator\_

y\_pred = best\_model.predict(X\_test)

**Ensemble Models** (page 222 of book as well)

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

from sklearn.datasets import load\_iris

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.svm import SVC

from sklearn.ensemble import VotingClassifier

from sklearn.metrics import accuracy\_score

# Load a sample dataset for demonstration purposes

iris = load\_iris()

X, y = iris.data, 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)

# Define individual classifiers

clf1 = RandomForestClassifier(n\_estimators=50, random\_state=42)

clf2 = GradientBoostingClassifier(n\_estimators=50, random\_state=42)

clf3 = SVC(probability=True, kernel='linear', random\_state=42)

# Create a VotingClassifier with soft voting

# Set `voting='soft'` to use probabilities for decision

voting\_clf = VotingClassifier(estimators=[('rf', clf1), ('gb', clf2), ('svc', clf3)], voting='soft')

# Fit the ensemble model on the training data

voting\_clf.fit(X\_train, y\_train)

# Predictions on the test set

y\_pred = voting\_clf.predict(X\_test)

# Accuracy score on the test set

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

print("Accuracy:", accuracy)

# Get probabilities for each class on the test set

probs = mv\_clf.predict\_proba(X\_test)

# Evaluate using ROC curve

fpr, tpr, thresholds = roc\_curve(y\_test, probs[:, 1])

roc\_auc = auc(fpr, tpr)

# Plot ROC curve

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

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC)')

plt.legend(loc='lower right')

plt.show()

**bagging**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

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

from sklearn.metrics import accuracy\_score

# Create a synthetic dataset for demonstration

X, y = make\_classification(n\_samples=100, n\_features=2, n\_informative=2, n\_redundant=0,

n\_clusters\_per\_class=1, 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 base decision tree classifier

base\_classifier = DecisionTreeClassifier(random\_state=42)

# Create a BaggingClassifier with decision trees

bagging\_classifier = BaggingClassifier(base\_classifier, n\_estimators=10, random\_state=42)

# Fit the bagging classifier on the training data

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

# Make predictions on the test set

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

# Calculate and print the accuracy score

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

print(f"Accuracy Score: {accuracy:.2f}")

# Plot decision boundaries

def plot\_decision\_boundary(X, y, classifier, title):

h = .02 # step size in the mesh

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.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

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

Z = Z.reshape(xx.shape)

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

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

plt.title(title)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()

# Plot decision boundaries for the base classifier

plot\_decision\_boundary(X\_train, y\_train, base\_classifier, title='Decision Boundaries - Base Classifier')

# Plot decision boundaries for the bagging classifier

plot\_decision\_boundary(X\_train, y\_train, bagging\_classifier, title='Decision Boundaries - Bagging Classifier')

**Boosting**

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

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

from sklearn.metrics import accuracy\_score

# Create a synthetic dataset for demonstration

X, y = make\_classification(n\_samples=100, n\_features=2, n\_informative=2, n\_redundant=0,

n\_clusters\_per\_class=1, 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 base decision tree classifier

base\_classifier = DecisionTreeClassifier(max\_depth=1, random\_state=42)

# Create an AdaBoost classifier with decision trees as base classifiers

adaboost\_classifier = AdaBoostClassifier(base\_classifier, n\_estimators=50, random\_state=42)

# Fit the AdaBoost classifier on the training data

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

# Make predictions on the test set

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

# Calculate and print the accuracy score

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

print(f"Accuracy Score: {accuracy:.2f}")

# You can also access the individual weak classifiers (base classifiers)

print("Individual Weak Classifiers:", adaboost\_classifier.estimators\_)

# Note: AdaBoost often performs well with weak learners, such as shallow decision trees.

**K-Means**

pip install yellowbrick

from yellowbrick.cluster import KElbowVisualizer

from sklearn.cluster import KMeans

Elbow\_Curve = KElbowVisualizer(KMeans(), k=(2,10))

Elbow\_Curve.fit(pca\_df)

Elbow\_Curve.show()

# Apply K-means clustering with K-means++ initialization

kmeans = KMeans(n\_clusters=4, init='k-means++', random\_state=42)

pca\_df['KM\_labels'] = kmeans.fit\_predict(pca\_df)

# Visualize the clusters

plt.scatter(pca\_df['PC1'], pca\_df['PC2'], c=pca\_df['KM\_labels'], s=50, cmap='viridis')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], marker='X', s=200, c='red', label='Centroids')

plt.title('K-means Clustering Results with K-means++ Initialization')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.show()

pca\_df['KM\_labels']

print("Unique values in 'KM\_labels':", pca\_df['KM\_labels'].unique())

cluster\_stats = pca\_df.groupby('KM\_labels')

cluster\_sizes = pca\_df['KM\_labels'].value\_counts()

print("Cluster Sizes:")

print(cluster\_sizes)

from sklearn.metrics import silhouette\_samples, silhouette\_score

silhouette\_avg = silhouette\_score(pca\_df[['PC1', 'PC2']], pca\_df['KM\_labels'])

print(f"Average Silhouette Score: {silhouette\_avg}")

kmeans\_results = []

for k in range(2, 6):

kmeans = KMeans(n\_clusters=k, init='k-means++', random\_state=42)

pca\_df['KM\_labels'] = kmeans.fit\_predict(pca\_df)

silhouette\_score\_value = silhouette\_score(pca\_df, pca\_df['KM\_labels'].astype(int))

kmeans\_results.append({'Number of Clusters': k, 'Silhouette Score': silhouette\_score\_value})

result\_df = pd.DataFrame(kmeans\_results)

print(result\_df)

**Agglomerative**

from scipy.cluster.hierarchy import dendrogram, linkage

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

linkage\_ = linkage(X\_scaled, method='ward')

dendrogram\_ = dendrogram(linkage\_)

from sklearn.cluster import AgglomerativeClustering

from scipy.cluster.hierarchy import dendrogram, linkage

# Apply hierarchical clustering

hc = AgglomerativeClustering(n\_clusters=3, linkage='ward', metric='euclidean')

pca\_df['HC\_labels'] = hc.fit\_predict(pca\_df[['PC1', 'PC2']])

# Plot the hierarchical clustering dendrogram

linked = linkage(pca\_df[['PC1', 'PC2']], method='ward')

dendrogram(linked, orientation='top', distance\_sort='descending', show\_leaf\_counts=True)

plt.title('Hierarchical Clustering Dendrogram')

plt.xlabel('Sample Index')

plt.ylabel('Distance')

plt.show()

plt.scatter(pca\_df['PC1'], pca\_df['PC2'], c= pca\_df['HC\_labels'], s=50, cmap='viridis')

plt.title('Hierarchical Clustering Results')

plt.xlabel('PC 1')

plt.ylabel('PC 2')

plt.show()

from sklearn.metrics import silhouette\_samples, silhouette\_score

silhouette\_avg = silhouette\_score(pca\_df[['PC1', 'PC2']], pca\_df['HC\_labels'])

print(f"Average Silhouette Score: {silhouette\_avg}")

#testing multiple combinations of linkage and metrics to find optimal silhouette score

data = pca\_df[['PC1', 'PC2']]

n\_clusters\_range = range(2, 6)

linkage\_methods = ['complete', 'average']

distance\_metrics = ['euclidean', 'manhattan', 'cosine']

results = []

for n\_clusters, linkage\_method, distance\_metric in product(n\_clusters\_range, linkage\_methods, distance\_metrics):

hc = AgglomerativeClustering(n\_clusters=n\_clusters, linkage=linkage\_method, metric=distance\_metric)

cluster\_labels = hc.fit\_predict(data)

silhouette\_avg = silhouette\_score(data, cluster\_labels)

results.append({

'n\_clusters': n\_clusters,

'linkage\_method': linkage\_method,

'distance\_metric': distance\_metric,

'silhouette\_score': silhouette\_avg

})

results\_df = pd.DataFrame(results)

print(results\_df)

**DBScan**

# Apply DBSCAN clustering

from sklearn.cluster import DBSCAN

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

pca\_df['DB\_labels']= dbscan.fit\_predict(pca\_df)

# Visualize the clusters

plt.scatter(pca\_df['PC1'], pca\_df['PC2'], c=pca\_df['DB\_labels'], s=50, cmap='viridis')

plt.title('DBSCAN Clustering Results')

plt.xlabel('PC 1')

plt.ylabel('PC 2')

plt.show()

from sklearn.metrics import silhouette\_samples, silhouette\_score

silhouette\_avg = silhouette\_score(pca\_df[['PC1', 'PC2']], pca\_df['HC\_labels'])

print(f"Average Silhouette Score: {silhouette\_avg}")

**Perceptron**

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import Perceptron

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

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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)

# Preprocess the data (standard scaling in this case)

scaler = StandardScaler()

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

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

# Experiment with different hyperparameter values

alpha\_values = [0.001, 0.01, 0.1]

max\_iter\_values = [100, 500, 1000]

for alpha in alpha\_values:

for max\_iter in max\_iter\_values:

print(f"\nHyperparameters: alpha={alpha}, max\_iter={max\_iter}")

# Initialize the Perceptron with hyperparameters

perceptron = Perceptron(alpha=alpha, max\_iter=max\_iter, random\_state=42)

# Fit the model on the scaled training data

perceptron.fit(X\_train\_scaled, y\_train)

# Make predictions on the scaled test data

y\_pred = perceptron.predict(X\_test\_scaled)

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)

**SVM**  
import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

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

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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)

# Preprocess the data (standard scaling in this case)

scaler = StandardScaler()

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

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

# Experiment with different hyperparameter values

C\_values = [0.1, 1, 10]

kernel\_values = ['linear', 'rbf', 'poly']

for C in C\_values:

for kernel in kernel\_values:

print(f"\nHyperparameters: C={C}, kernel={kernel}")

# Initialize the SVM classifier with hyperparameters

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

# Fit the model on the scaled training data

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

# Make predictions on the scaled test data

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

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)

**Logistic Regression**

import pandas as pd

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, classification\_report, confusion\_matrix

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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)

# Preprocess the data (standard scaling in this case)

scaler = StandardScaler()

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

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

# Experiment with different hyperparameter values

C\_values = [0.1, 1, 10]

solver\_values = ['liblinear', 'lbfgs', 'newton-cg']

for C in C\_values:

for solver in solver\_values:

print(f"\nHyperparameters: C={C}, solver={solver}")

# Initialize the Logistic Regression classifier with hyperparameters

logistic\_regression = LogisticRegression(C=C, solver=solver, random\_state=42)

# Fit the model on the scaled training data

logistic\_regression.fit(X\_train\_scaled, y\_train)

# Make predictions on the scaled test data

y\_pred = logistic\_regression.predict(X\_test\_scaled)

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)

**Decision trees**

import pandas as pd

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

from sklearn.tree import DecisionTreeClassifier

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

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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)

# Experiment with different hyperparameter values

max\_depth\_values = [None, 5, 10]

min\_samples\_split\_values = [2, 5, 10]

for max\_depth in max\_depth\_values:

for min\_samples\_split in min\_samples\_split\_values:

print(f"\nHyperparameters: max\_depth={max\_depth}, min\_samples\_split={min\_samples\_split}")

# Initialize the Decision Tree classifier with hyperparameters

decision\_tree\_classifier = DecisionTreeClassifier(max\_depth=max\_depth, min\_samples\_split=min\_samples\_split, random\_state=42)

# Fit the model on the training data

decision\_tree\_classifier.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = decision\_tree\_classifier.predict(X\_test)

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)

**KNN**  
import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

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

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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)

# Preprocess the data (standard scaling in this case)

scaler = StandardScaler()

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

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

# Experiment with different hyperparameter values

n\_neighbors\_values = [3, 5, 7]

weights\_values = ['uniform', 'distance']

for n\_neighbors in n\_neighbors\_values:

for weights in weights\_values:

print(f"\nHyperparameters: n\_neighbors={n\_neighbors}, weights={weights}")

# Initialize the KNN classifier with hyperparameters

knn\_classifier = KNeighborsClassifier(n\_neighbors=n\_neighbors, weights=weights)

# Fit the model on the scaled training data

knn\_classifier.fit(X\_train\_scaled, y\_train)

# Make predictions on the scaled test data

y\_pred = knn\_classifier.predict(X\_test\_scaled)

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)

**Grid Search CV**

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import Pipeline

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

# Assume 'df' is your DataFrame with features and target variable

# Replace 'target\_column' with the actual name of your target column

X = df.drop('target\_column', axis=1)

y = df['target\_column']

# 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 pipeline with StandardScaler, PCA, and a classifier

# Experiment with different classifiers and hyperparameter grids

pipeline = Pipeline([

('scaler', StandardScaler()),

('pca', PCA()),

('classifier', RandomForestClassifier()) # Change this to your desired classifier

])

# Define hyperparameter grids for GridSearchCV

param\_grid = {

'pca\_\_n\_components': [None, 5, 10], # Number of components for PCA

'classifier\_\_n\_estimators': [50, 100, 200], # Number of trees for RandomForestClassifier

'classifier\_\_max\_depth': [None, 5, 10], # Max depth for RandomForestClassifier

# Add hyperparameters for other classifiers as needed

}

# Initialize GridSearchCV

grid\_search = GridSearchCV(pipeline, param\_grid, cv=5, scoring='accuracy')

# Fit the model on the training data

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

# Make predictions on the test data

y\_pred = grid\_search.predict(X\_test)

# Evaluate the model

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

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

classification\_rep = classification\_report(y\_test, y\_pred)

print("Best Hyperparameters:", grid\_search.best\_params\_)

print("Accuracy:", accuracy)

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(classification\_rep)