1. **ID3 ALGORITHM**

import numpy as np

def entropy(y):

classes, counts = np.unique(y, return\_counts=True)

probabilities = counts / len(y)

entropy = -np.sum(probabilities \* np.log2(probabilities))

return entropy

def information\_gain(X, y, feature\_idx):

total\_entropy = entropy(y)

values, counts = np.unique(X[:, feature\_idx], return\_counts=True)

weighted\_entropy = np.sum([(counts[i] / np.sum(counts)) \* entropy(y[X[:, feature\_idx] == values[i]]) for i in range(len(values))])

information\_gain = total\_entropy - weighted\_entropy

return information\_gain

def id3(X, y, features):

if len(np.unique(y)) == 1:

return np.unique(y)[0]

if len(features) == 0:

return np.bincount(y).argmax()

information\_gains = [information\_gain(X, y, feature\_idx) for feature\_idx in range(X.shape[1])]

best\_feature\_idx = np.argmax(information\_gains)

best\_feature = features[best\_feature\_idx]

tree = {best\_feature: {}}

remaining\_features = [feature for i, feature in enumerate(features) if i != best\_feature\_idx]

values, counts = np.unique(X[:, best\_feature\_idx], return\_counts=True)

for value, count in zip(values, counts):

sub\_X = X[X[:, best\_feature\_idx] == value]

sub\_y = y[X[:, best\_feature\_idx] == value]

tree[best\_feature][value] = id3(sub\_X, sub\_y, remaining\_features)

return tree

1. **Random forest algorithm.**

import numpy as np

from sklearn.tree import DecisionTreeClassifier

class RandomForestClassifier:

def \_\_init\_\_(self, n\_estimators=10, max\_depth=None, random\_state=None):

self.n\_estimators = n\_estimators

self.max\_depth = max\_depth

self.random\_state = random\_state

self.estimators = []

def fit(self, X, y):

np.random.seed(self.random\_state)

for \_ in range(self.n\_estimators):

sample\_indices = np.random.choice(len(X), size=len(X), replace=True)

X\_sampled = X[sample\_indices]

y\_sampled = y[sample\_indices]

tree = DecisionTreeClassifier(max\_depth=self.max\_depth)

tree.fit(X\_sampled, y\_sampled)

self.estimators.append(tree)

def predict(self, X):

predictions = np.zeros((len(X), len(self.estimators)))

for i, tree in enumerate(self.estimators):

predictions[:, i] = tree.predict(X)

return np.apply\_along\_axis(lambda x: np.bincount(x).argmax(), axis=1, arr=predictions)

1. **SVM classifier**

from sklearn.svm import SVC

from sklearn.preprocessing import StandardScaler

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

from sklearn.metrics import accuracy\_score

# Assuming you have your input features in X and corresponding labels in y

# Split the data into training and test sets

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

# Scale the features using StandardScaler

scaler = StandardScaler()

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

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

# Create an instance of the SVC classifier

svm\_classifier = SVC()

# Train the SVM classifier

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

# Make predictions on the test set

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

# Calculate the accuracy of the classifier

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

print("Accuracy:", accuracy)

1. **Candidate-Elimination algorithm**

import numpy as np

def initialize\_bounds(n\_features):

S = [('?',) \* n\_features]

G = [('',) \* n\_features]

return S, G

def consistent(h, instance):

for x, y in zip(h, instance):

if x == '?' or x == y:

continue

else:

return False

return True

def generalize\_S(h, instance):

new\_h = list(h)

for i in range(len(h)):

if h[i] == '?':

new\_h[i] = instance[i]

elif h[i] != instance[i]:

new\_h[i] = '?'

return tuple(new\_h)

def specialize\_G(S, G, instance):

new\_G = []

for g in G:

if not consistent(g, instance):

continue

for i in range(len(g)):

if g[i] != '?' and g[i] != instance[i]:

new\_g = list(g)

new\_g[i] = '?'

new\_G.append(tuple(new\_g))

if g not in S:

new\_G.append(g)

return new\_G

def candidate\_elimination(X, y):

n\_features = X.shape[1]

S, G = initialize\_bounds(n\_features)

for i, instance in enumerate(X):

if y[i] == 1:

# Positive instance

G = [g for g in G if consistent(g, instance)]

S = generalize\_S(S, instance)

S = [s for s in S if not any(consistent(s, x) and not consistent(s, y) for y in S)]

G = specialize\_G(S, G, instance)

else:

# Negative instance

S = [s for s in S if not consistent(s, instance)]

G = [g for g in G if any(g[i] == '?' or g[i] != instance[i] for i in range(n\_features))]

return S, G

1. **K-means algorithm**

import numpy as np

def initialize\_centers(X, K):

n\_samples, n\_features = X.shape

centers = np.zeros((K, n\_features))

random\_indices = np.random.choice(n\_samples, size=K, replace=False)

centers = X[random\_indices]

return centers

def assign\_clusters(X, centers):

n\_samples = X.shape[0]

clusters = np.zeros(n\_samples)

for i in range(n\_samples):

distances = np.linalg.norm(X[i] - centers, axis=1)

cluster = np.argmin(distances)

clusters[i] = cluster

return clusters

def update\_centers(X, clusters, K):

n\_features = X.shape[1]

centers = np.zeros((K, n\_features))

for k in range(K):

cluster\_samples = X[clusters == k]

centers[k] = np.mean(cluster\_samples, axis=0)

return centers

def k\_means(X, K, max\_iterations=100):

centers = initialize\_centers(X, K)

for \_ in range(max\_iterations):

prev\_centers = np.copy(centers)

clusters = assign\_clusters(X, centers)

centers = update\_centers(X, clusters, K)

if np.all(prev\_centers == centers):

break

return centers, clusters

# Example usage:

# Assuming you have your input data in X

K = 3 # Number of clusters

centers, clusters = k\_means(X, K)

print("Cluster centers:", centers)

print("Assigned clusters:", clusters)

1. **Locally Weighted Regression algorithm**

import numpy as np

def locally\_weighted\_regression(X\_train, y\_train, x\_test, tau):

m = X\_train.shape[0]

n = X\_train.shape[1]

# Add a column of ones to X\_train for the bias term

X\_train\_bias = np.c\_[np.ones((m, 1)), X\_train]

# Add a column of ones to x\_test for the bias term

x\_test\_bias = np.c\_[1, x\_test]

# Calculate the weights for each training example based on the given test example

weights = np.exp(-np.sum((X\_train - x\_test) \*\* 2, axis=1) / (2 \* tau \*\* 2))

# Calculate the weighted least squares solution

theta = np.linalg.inv(X\_train\_bias.T.dot(np.diag(weights)).dot(X\_train\_bias)).dot(X\_train\_bias.T).dot(np.diag(weights)).dot(y\_train)

# Predict the target value for the test example

y\_pred = x\_test\_bias.dot(theta)

return y\_pred

# Example usage:

# Assuming you have your training data in X\_train and corresponding targets in y\_train

# Assuming you have a test example in x\_test

tau = 0.5 # Bandwidth parameter

y\_pred = locally\_weighted\_regression(X\_train, y\_train, x\_test, tau)

print("Predicted target value:", y\_pred)