1.Decision tree

import pandas as pd

import numpy as np

# Define a Node class to represent each node in the decision tree

class Node:

def \_init\_(self, feature=None, threshold=None, left=None, right=None, \*, value=None):

self.feature = feature # Index of the feature to split on

self.threshold = threshold # Threshold value for the feature

self.left = left # Left child node

self.right = right # Right child node

self.value = value # Value if the node is a leaf

def is\_leaf\_node(self):

return self.value is not None

# Define the DecisionTreeClassifier class

class DecisionTreeClassifier:

def \_init\_(self, max\_depth=100):

self.max\_depth = max\_depth

self.root = None

def fit(self, X, y):

self.root = self.\_grow\_tree(X, y)

def predict(self, X):

return np.array([self.\_traverse\_tree(x, self.root) for x in X])

def \_grow\_tree(self, X, y, depth=0):

n\_samples, n\_features = X.shape

n\_labels = len(np.unique(y))

# Stopping criteria

if depth >= self.max\_depth or n\_labels == 1 or n\_samples < 2:

leaf\_value = self.\_most\_common\_label(y)

return Node(value=leaf\_value)

rnd\_feats = np.random.choice(n\_features, n\_features, replace=False)

# Find the best split

best\_feat, best\_thresh = self.\_best\_criteria(X, y, rnd\_feats)

# Grow the children recursively

left\_idx, right\_idx = self.\_split(X[:, best\_feat], best\_thresh)

left = self.\_grow\_tree(X[left\_idx, :], y[left\_idx], depth + 1)

right = self.\_grow\_tree(X[right\_idx, :], y[right\_idx], depth + 1)

return Node(best\_feat, best\_thresh, left, right)

def \_best\_criteria(self, X, y, features):

best\_gain = -1

split\_idx, split\_thresh = None, None

for feat in features:

X\_column = X[:, feat]

thresholds = np.unique(X\_column)

for threshold in thresholds:

gain = self.\_information\_gain(y, X\_column, threshold)

if gain > best\_gain:

best\_gain = gain

split\_idx = feat

split\_thresh = threshold

return split\_idx, split\_thresh

def \_information\_gain(self, y, X\_column, split\_thresh):

# Parent Gini impurity

parent\_gini = self.\_gini(y)

# Generate split

left\_idx, right\_idx = self.\_split(X\_column, split\_thresh)

if len(left\_idx) == 0 or len(right\_idx) == 0:

return 0

# Weighted avg Gini of children

n = len(y)

n\_l, n\_r = len(left\_idx), len(right\_idx)

e\_l, e\_r = self.\_gini(y[left\_idx]), self.\_gini(y[right\_idx])

child\_gini = (n\_l / n) \* e\_l + (n\_r / n) \* e\_r

# Information gain is parent impurity minus child impurity

ig = parent\_gini - child\_gini

return ig

def \_gini(self, y):

m = len(y)

return 1.0 - sum((np.sum(y == c) / m) \*\* 2 for c in np.unique(y))

def \_split(self, X\_column, split\_thresh):

left\_idx = np.argwhere(X\_column <= split\_thresh).flatten()

right\_idx = np.argwhere(X\_column > split\_thresh).flatten()

return left\_idx, right\_idx

def \_traverse\_tree(self, x, node):

if node.is\_leaf\_node():

return node.value

if x[node.feature] <= node.threshold:

return self.\_traverse\_tree(x, node.left)

return self.\_traverse\_tree(x, node.right)

def \_most\_common\_label(self, y):

return np.bincount(y).argmax()

# Step 1: Create the CSV file

data = {

'Day': ['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10', 'D11', 'D12', 'D13', 'D14'],

'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain', 'Overcast', 'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast', 'Overcast', 'Rain'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],

'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Strong'],

'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']

}

df = pd.DataFrame(data)

csv\_file\_path = 'tennis.csv'

df.to\_csv(csv\_file\_path, index=False)

# Step 2: Load the CSV file and apply the Decision Tree Algorithm

# Load dataset

df = pd.read\_csv(csv\_file\_path)

# Convert categorical variables to numeric

df['Outlook'] = df['Outlook'].map({'Sunny': 0, 'Overcast': 1, 'Rain': 2})

df['Temperature'] = df['Temperature'].map({'Hot': 0, 'Mild': 1, 'Cool': 2})

df['Humidity'] = df['Humidity'].map({'High': 0, 'Normal': 1})

df['Wind'] = df['Wind'].map({'Weak': 0, 'Strong': 1})

df['PlayTennis'] = df['PlayTennis'].map({'No': 0, 'Yes': 1})

# Split the data into features and target

X = df.drop(['Day', 'PlayTennis'], axis=1).values

y = df['PlayTennis'].values

# Initialize and train the Decision Tree Classifier

clf = DecisionTreeClassifier(max\_depth=3)

clf.fit(X, y)

# Make predictions

y\_pred = clf.predict(X)

# Evaluate the model

accuracy = np.sum(y == y\_pred) / len(y)

print(f"Accuracy: {accuracy}")

2.K means

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Create the CSV file

data = {

'Individual': [1, 2, 3, 4, 5, 6, 7],

'Variable 1': [1.0, 1.5, 3.0, 5.0, 3.5, 4.5, 3.5],

'Variable 2': [1.0, 2.0, 4.0, 7.0, 5.0, 5.0, 4.5]

}

df = pd.DataFrame(data)

csv\_file\_path = 'kmeans\_data.csv'

df.to\_csv(csv\_file\_path, index=False)

# Step 2: Implement K-means Clustering Algorithm from Scratch

class KMeans:

def \_init\_(self, K=2, max\_iters=100):

self.K = K

self.max\_iters = max\_iters

self.centroids = None

def fit(self, X):

# Randomly initialize centroids

self.centroids = X[np.random.choice(X.shape[0], self.K, replace=False)]

for \_ in range(self.max\_iters):

# Assign clusters

clusters = self.\_assign\_clusters(X)

# Calculate new centroids

new\_centroids = self.\_calculate\_centroids(X, clusters)

# Check for convergence

if np.all(self.centroids == new\_centroids):

break

self.centroids = new\_centroids

def predict(self, X):

return self.\_assign\_clusters(X)

def \_assign\_clusters(self, X):

distances = np.array([[np.linalg.norm(x - centroid) for centroid in self.centroids] for x in X])

return np.argmin(distances, axis=1)

def \_calculate\_centroids(self, X, clusters):

return np.array([X[clusters == k].mean(axis=0) for k in range(self.K)])

# Load the CSV file

df = pd.read\_csv(csv\_file\_path)

# Extract features

X = df[['Variable 1', 'Variable 2']].values

# Initialize and fit the KMeans model

kmeans = KMeans(K=2, max\_iters=100)

kmeans.fit(X)

# Predict the clusters

clusters = kmeans.predict(X)

# Add the cluster labels to the dataframe

df['Cluster'] = clusters

# Print the resulting clusters

print(df)

# Step 3: Visualize the clusters

def plot\_clusters(df, kmeans):

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

# Plot each cluster

for cluster in range(kmeans.K):

cluster\_points = df[df['Cluster'] == cluster]

plt.scatter(cluster\_points['Variable 1'], cluster\_points['Variable 2'], label=f'Cluster {cluster}')

# Plot centroids

for centroid in kmeans.centroids:

plt.scatter(\*centroid, s=200, marker='X', c='black')

plt.xlabel('Variable 1')

plt.ylabel('Variable 2')

plt.title('K-means Clustering')

plt.legend()

plt.show()

plot\_clusters(df, kmeans)

3.linear regression

import numpy as np

# Given data points

x = np.array([0, 1, 2, 3, 4])

y = np.array([2, 3, 5, 4, 6])

# a) Find the linear regression line y = ax + b

# Calculate the means of x and y

x\_mean = np.mean(x)

y\_mean = np.mean(y)

# Calculate the terms needed for the numerator and denominator of 'a'

numerator = 0

denominator = 0

for i in range(len(x)):

numerator += (x[i] - x\_mean) \* (y[i] - y\_mean)

denominator += (x[i] - x\_mean) \*\* 2

# Calculate 'a' and 'b'

a = numerator / denominator

b = y\_mean - a \* x\_mean

print(f"Linear regression line: y = {a:.2f}x + {b:.2f}")

# b) Estimate the value of y when x = 10

x\_new = 10

y\_new = a \* x\_new + b

print(f"Estimated value of y when x = 10: {y\_new:.2f}")

# c) Calculate the error

# Error is the sum of squared differences between actual and predicted values

error = 0

for i in range(len(x)):

y\_pred = a \* x[i] + b

error += (y[i] - y\_pred) \*\* 2

print(f"Sum of squared errors: {error:.2f}")

# Additionally, to calculate mean squared error (MSE) for better error representation

mse = error / len(x)

print(f"Mean squared error: {mse:.2f}")

4.Clustering

import numpy as np

import matplotlib.pyplot as plt

# Given points

points = np.array([

[2, 10], # A1

[2, 5], # A2

[8, 4], # A3

[5, 8], # A4

[7, 5], # A5

[6, 4], # A6

[1, 2], # A7

[4, 9] # A8

])

# Initial cluster centers

initial\_centers = np.array([

[2, 10], # Center for cluster 1 (A1)

[5, 8], # Center for cluster 2 (A4)

[1, 2] # Center for cluster 3 (A7)

])

# Function to compute the Euclidean distance

def euclidean\_distance(a, b):

return np.sqrt(np.sum((a - b) \*\* 2))

# Function to assign points to the nearest cluster center

def assign\_clusters(points, centers):

clusters = {}

for i in range(len(centers)):

clusters[i] = []

for point in points:

distances = [euclidean\_distance(point, center) for center in centers]

cluster = np.argmin(distances)

clusters[cluster].append(point)

return clusters

# Function to recalculate the cluster centers

def recalculate\_centers(clusters):

new\_centers = []

for cluster in clusters.values():

new\_center = np.mean(cluster, axis=0)

new\_centers.append(new\_center)

return new\_centers

# K-means clustering algorithm

def kmeans(points, initial\_centers, max\_iterations=100):

centers = initial\_centers

for \_ in range(max\_iterations):

clusters = assign\_clusters(points, centers)

new\_centers = recalculate\_centers(clusters)

if np.allclose(centers, new\_centers):

break

centers = new\_centers

return centers, clusters

# Run the K-means algorithm

final\_centers, clusters = kmeans(points, initial\_centers)

print("Final cluster centers:")

for i, center in enumerate(final\_centers):

print(f"Cluster {i+1} center: {center}")

print("\nCluster assignments:")

for i, cluster in clusters.items():

print(f"Cluster {i+1}: {cluster}")

# Convert final\_centers to a numpy array

final\_centers = np.array(final\_centers)

# Plot the results

colors = ['r', 'b', 'g']

labels = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8']

for i, cluster in clusters.items():

cluster\_points = np.array(cluster)

plt.scatter(cluster\_points[:, 0], cluster\_points[:, 1], color=colors[i], label=f'Cluster {i+1}')

for point in cluster\_points:

plt.text(point[0], point[1], labels[np.where((points == point).all(axis=1))[0][0]], fontsize=12)

plt.scatter(final\_centers[:, 0], final\_centers[:, 1], color='k', marker='x', s=100, label='Centers')

plt.xlabel('x')

plt.ylabel('y')

plt.legend()

plt.title('K-means Clustering')

plt.show()

5.statistical measures

import re

from datetime import datetime

# Dataset

data = [

[1, 21, "1L", "Male", "31.05.1992", "No"],

[2, 35, "1,00,000", "Male", "10-05-2002", "Yes"],

[3, 26, "45000", "Male", "Aug 5, 2000", "Yes"],

[4, 45, "", "Male", " ", "No"],

[5, 67, "10,000", "Female", "31.03.1986", "Yes"],

[6, 32, "10000", "Female", "10/5/1987", "Yes"],

[7, 32, "5$", "Female", "31.05.1992", "Yes"],

[8, 31, "5 Dollars", "Male", "10-05-2002", "No"],

[9, 10, "10,000", "Female", "Aug 5, 2000", "Yes"],

[10, 42, "15000", "Female", "Sep 12'2000", "Yes"],

[11, "", "25,000", "Female", "31.03.1986", "Yes"],

[12, 32, "35000", "Male", "10/5/1987", "No"],

[13, 35, "150000", "Female", "Sep 12'2000", "Yes"],

[14, 35, "35000", "Male", "31.03.1986", "No"]

]

# Function to clean income

def clean\_income(income):

if income == "":

return 0

income = re.sub(r'[^\d]', '', income)

return int(income)

# Function to clean DoB

def clean\_dob(dob):

if dob.strip() == "":

return None

dob = re.sub(r'[^\w\s]', ' ', dob)

try:

return datetime.strptime(dob, "%d %m %Y").strftime("%Y-%m-%d")

except ValueError:

return None

# Clean data

for row in data:

row[2] = clean\_income(row[2])

row[4] = clean\_dob(row[4])

if row[1] == "":

row[1] = 0 # Assuming 0 for missing age

# Helper functions for statistical measures

def mean(values):

total = sum(values)

count = len(values)

return total / count if count != 0 else 0

def median(values):

values = sorted(values)

n = len(values)

mid = n // 2

if n % 2 == 0:

return (values[mid - 1] + values[mid]) / 2

else:

return values[mid]

def mode(values):

frequency = {}

for value in values:

frequency[value] = frequency.get(value, 0) + 1

max\_freq = max(frequency.values())

modes = [key for key, value in frequency.items() if value == max\_freq]

return modes

def standard\_deviation(values):

avg = mean(values)

variance = mean([(x - avg) \*\* 2 for x in values])

return variance \*\* 0.5

def gender\_count(data):

count = {'Male': 0, 'Female': 0}

for row in data:

count[row[3]] += 1

return count

# Extract columns for statistical measures

ages = [row[1] for row in data if row[1] != 0]

incomes = [row[2] for row in data if row[2] != 0]

# Calculate statistical measures

mean\_age = mean(ages)

median\_age = median(ages)

mode\_income = mode(incomes)

std\_dev\_income = standard\_deviation(incomes)

gender\_distribution = gender\_count(data)

# Print results

print(f"Mean Age: {mean\_age}")

print(f"Median Age: {median\_age}")

print(f"Mode of Income: {mode\_income}")

print(f"Standard Deviation of Income: {std\_dev\_income}")

print(f"Gender Count: {gender\_distribution}")

# Print cleaned data

print("\nCleaned Data:")

for row in data:

print(row)

6.K means

import numpy as np

import matplotlib.pyplot as plt

# Initial centers

centers = np.array([[2, 2], [1, 1]])

# Points

points = np.array([[2, 2], [3, 2], [1, 1], [3, 1], [1.5, 0.5]])

# Function to compute the distance between two points

def distance(a, b):

return np.sqrt(np.sum((a - b) \*\* 2))

# K-Means algorithm

def k\_means(points, centers):

# Assign points to the nearest center

clusters = {i: [] for i in range(len(centers))}

for point in points:

distances = [distance(point, center) for center in centers]

closest\_center = np.argmin(distances)

clusters[closest\_center].append(point)

# Update the centers

new\_centers = np.array([np.mean(clusters[i], axis=0) for i in range(len(centers))])

return clusters, new\_centers

# Plotting function

def plot\_clusters(clusters, centers):

colors = ['r', 'g', 'b', 'c', 'm']

for idx, cluster in clusters.items():

cluster = np.array(cluster)

plt.scatter(cluster[:, 0], cluster[:, 1], c=colors[idx], label=f'Cluster {idx+1}')

plt.scatter(centers[:, 0], centers[:, 1], c='black', marker='x', label='Centers')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.legend()

plt.title('K-Means Clustering')

plt.show()

# Perform K-Means clustering

clusters, new\_centers = k\_means(points, centers)

# Plot initial clusters

plot\_clusters(clusters, new\_centers)

# Repeat until centers do not change

iteration = 1

while not np.array\_equal(centers, new\_centers):

print(f'Iteration {iteration}: Centers updated to {new\_centers}')

centers = new\_centers

clusters, new\_centers = k\_means(points, centers)

iteration += 1

# Final plot

print(f'Final centers: {new\_centers}')

plot\_clusters(clusters, new\_centers)

7.Apriori algo

# Dataset

transactions = {

'T1': {'HotDogs', 'Buns', 'Ketchup'},

'T2': {'HotDogs', 'Buns'},

'T3': {'HotDogs', 'Coke', 'Chips'},

'T4': {'Chips', 'Coke'},

'T5': {'Chips', 'Ketchup'},

'T6': {'HotDogs', 'Coke', 'Chips'}

}

# Parameters

support\_threshold = 33.34 / 100

confidence\_threshold = 60 / 100

# Function to calculate support

def calculate\_support(itemset, transactions):

count = 0

for transaction in transactions.values():

if itemset.issubset(transaction):

count += 1

return count / len(transactions)

# Function to generate combinations of a certain length

def generate\_combinations(items, length):

if length == 0:

return [set()]

elif length == 1:

return [{item} for item in items]

else:

combinations = []

for i in range(len(items)):

for subset in generate\_combinations(items[i+1:], length-1):

combinations.append({items[i]} | subset)

return combinations

# Apriori algorithm

def apriori(transactions, support\_threshold):

items = set(item for transaction in transactions.values() for item in transaction)

candidates = [frozenset([item]) for item in items]

frequent\_itemsets = []

k = 1

while candidates:

print(f"Scanning for itemsets of length {k}")

candidate\_supports = {item: calculate\_support(item, transactions) for item in candidates}

frequent\_items = {item for item, support in candidate\_supports.items() if support >= support\_threshold}

frequent\_itemsets.extend(frequent\_items)

print(f"Candidates: {candidates}")

print(f"Frequent itemsets: {frequent\_items}")

k += 1

candidates = generate\_combinations(list(set(item for itemset in frequent\_items for item in itemset)), k)

candidates = [frozenset(candidate) for candidate in candidates]

return frequent\_itemsets

# Generate association rules

def generate\_rules(frequent\_itemsets, transactions, confidence\_threshold):

rules = []

for itemset in frequent\_itemsets:

itemset\_list = list(itemset)

for i in range(1, len(itemset\_list)):

for antecedent in generate\_combinations(itemset\_list, i):

antecedent = frozenset(antecedent)

consequent = itemset - antecedent

support = calculate\_support(itemset, transactions)

confidence = support / calculate\_support(antecedent, transactions)

if confidence >= confidence\_threshold:

rules.append((antecedent, consequent, confidence))

return rules

# Run Apriori algorithm

frequent\_itemsets = apriori(transactions, support\_threshold)

print("\nAll Frequent Itemsets:")

for itemset in frequent\_itemsets:

print(itemset)

# Generate and sort rules by confidence

rules = generate\_rules(frequent\_itemsets, transactions, confidence\_threshold)

rules.sort(key=lambda x: x[2], reverse=True)

print("\nStrong Association Rules (sorted by confidence):")

for antecedent, consequent, confidence in rules:

print(f"{set(antecedent)} -> {set(consequent)} (confidence: {confidence:.2f})")