from google.colab import files

uploaded=files.upload()

**Q1.**

import pandas as pd

import numpy as np

def equal\_width\_binning(data, num\_bins):

"""Performs equal-width binning on a pandas Series.

Args:

data: pandas Series containing the data to be binned.

num\_bins: The desired number of bins.

Returns:

A pandas Series with the binned data.

"""

# Calculate bin edges

min\_val = data.min()

max\_val = data.max()

bin\_width = (max\_val - min\_val) / num\_bins

bin\_edges = [min\_val + i \* bin\_width for i in range(num\_bins + 1)]

# Bin the data

binned\_data = pd.cut(data, bins=bin\_edges, labels=False, include\_lowest=True, duplicates='drop')

return binned\_data

def calculate\_entropy(data):

"""Calculates the entropy of a pandas Series.

Args:

data: pandas Series containing the data.

Returns:

The entropy of the data.

"""

# Calculate probabilities of each unique value

value\_counts = data.value\_counts(normalize=True)

probabilities = value\_counts.values

# Calculate entropy

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

return entropy

# --- Main part to read from CSV and calculate entropy ---

# Replace 'your\_csv\_file.csv' with the actual path to your CSV file

# Replace 'outcome\_column' with the name of the column containing your outcome variable

csv\_file\_path = 'Heart\_disease\_statlog.csv'

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

outcome\_column = 'target'

# Bin the outcome column if it's continuous:

df['binned\_outcome'] = equal\_width\_binning(df[outcome\_column], num\_bins=4)

# Calculate entropy:

entropy = calculate\_entropy(df['binned\_outcome'])

# Print the result

print(f"Entropy: {entropy}")

**OUTPUT**: Entropy: 0.9910760598382222

**Q2.**

import pandas as pd

import numpy as np

def calculate\_gini\_index(data, target\_column):

"""Calculates the Gini index for a dataset.

Args:

data: pandas DataFrame containing the data.

target\_column: The name of the column containing the target variable.

Returns:

The Gini index value.

"""

# Get the target variable values

target\_values = data[target\_column].unique()

# Calculate the total number of samples

total\_samples = len(data)

# Initialize Gini index to 0

gini\_index = 0

# Iterate through each target value

for value in target\_values:

# Calculate the proportion of samples with the current target value

proportion = len(data[data[target\_column] == value]) / total\_samples

# Update the Gini index

gini\_index += proportion \* (1 - proportion)

return gini\_index

# Assuming 'df' is your pandas DataFrame and 'target' is the name of the target column

gini\_index = calculate\_gini\_index(df, 'target')

# Print the result

print(f"Gini Index: {gini\_index}")

**OUTPUT:** Gini Index: 0.49382716049382713

**Q3.**

import pandas as pd

import numpy as np

def calculate\_entropy(data):

"""Calculates the entropy of a pandas Series."""

value\_counts = data.value\_counts(normalize=True)

probabilities = value\_counts.values

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

return entropy

def calculate\_information\_gain(data, feature\_column, target\_column):

"""Calculates the information gain for a given feature."""

total\_entropy = calculate\_entropy(data[target\_column])

# Calculate weighted average entropy for each feature value

weighted\_entropy = 0

for value in data[feature\_column].unique():

subset = data[data[feature\_column] == value]

weighted\_entropy += (len(subset) / len(data)) \* calculate\_entropy(subset[target\_column])

# Calculate information gain

information\_gain = total\_entropy - weighted\_entropy

return information\_gain

def find\_best\_root\_node(data, features, target\_column):

"""Finds the best feature for the root node using Information Gain."""

best\_feature = None

max\_information\_gain = -1 # Initialize with a negative value

for feature in features:

information\_gain = calculate\_information\_gain(data, feature, target\_column)

if information\_gain > max\_information\_gain:

max\_information\_gain = information\_gain

best\_feature = feature

return best\_feature

# Example Usage:

# Assuming 'df' is your pandas DataFrame, 'features' is a list of feature column names,

# and 'target\_column' is the name of the target variable column.

features = ['sex', 'cp', 'chol'] # Replace with your feature names

target\_column = 'target' # Replace with your target column name

best\_root\_feature = find\_best\_root\_node(df, features, target\_column)

print(f"Best Root Node Feature: {best\_root\_feature}")

**Q4.**

import pandas as pd

import numpy as np

import warnings

def binning(data, num\_bins=4, binning\_type='equal\_width'):

"""

Performs binning on a pandas Series.

Args:

data: pandas Series containing the data to be binned.

num\_bins: The desired number of bins (default is 4).

binning\_type: The type of binning to perform ('equal\_width' or 'equal\_frequency', default is 'equal\_width').

Returns:

A pandas Series with the binned data.

"""

if binning\_type == 'equal\_width':

# Calculate bin edges

min\_val = data.min()

max\_val = data.max()

bin\_width = (max\_val - min\_val) / num\_bins

bin\_edges = [min\_val + i \* bin\_width for i in range(num\_bins + 1)]

# Bin the data

binned\_data = pd.cut(data, bins=bin\_edges, labels=False, include\_lowest=True, duplicates='drop')

elif binning\_type == 'equal\_frequency':

# Bin the data using quantiles

binned\_data = pd.qcut(data, q=num\_bins, labels=False, duplicates='drop')

else:

raise ValueError("Invalid binning\_type. Choose 'equal\_width' or 'equal\_frequency'.")

return binned\_data

def calculate\_entropy(data):

"""Calculates the entropy of a pandas Series."""

value\_counts = data.value\_counts(normalize=True)

probabilities = value\_counts.values

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

return entropy

def calculate\_information\_gain(data, feature\_column, target\_column):

"""Calculates the information gain for a given feature."""

total\_entropy = calculate\_entropy(data[target\_column])

# Calculate weighted average entropy for each feature value

weighted\_entropy = 0

for value in data[feature\_column].unique():

subset = data[data[feature\_column] == value]

weighted\_entropy += (len(subset) / len(data)) \* calculate\_entropy(subset[target\_column])

# Calculate information gain

information\_gain = total\_entropy - weighted\_entropy

return information\_gain

def find\_best\_root\_node(data, features, target\_column):

"""Finds the best feature for the root node using Information Gain."""

best\_feature = None

max\_information\_gain = -1 # Initialize with a negative value

# Suppress DeprecationWarning

with warnings.catch\_warnings():

warnings.filterwarnings("ignore", category=DeprecationWarning)

for feature in features:

# Bin the feature if it's continuous

if data[feature].dtype == np.number:

data[feature] = binning(data[feature]) # Use default binning parameters

information\_gain = calculate\_information\_gain(data, feature, target\_column)

if information\_gain > max\_information\_gain:

max\_information\_gain = information\_gain

best\_feature = feature

return best\_feature

# Example Usage:

# Assuming 'df' is your pandas DataFrame, 'features' is a list of feature column names,

# and 'target\_column' is the name of the target variable column.

features = ['trestbps', 'chol', 'thalach'] # Replace with your feature names

target\_column = 'target' # Replace with your target column name

best\_root\_feature = find\_best\_root\_node(df, features, target\_column)

print(f"Best Root Node Feature: {best\_root\_feature}")

**Q5.**

import pandas as pd

import numpy as np

import warnings

def binning(data, num\_bins=4, binning\_type='equal\_width'):

"""Performs binning on a pandas Series."""

if binning\_type == 'equal\_width':

min\_val = data.min()

max\_val = data.max()

bin\_width = (max\_val - min\_val) / num\_bins

bin\_edges = [min\_val + i \* bin\_width for i in range(num\_bins + 1)]

binned\_data = pd.cut(data, bins=bin\_edges, labels=False, include\_lowest=True, duplicates='drop')

elif binning\_type == 'equal\_frequency':

binned\_data = pd.qcut(data, q=num\_bins, labels=False, duplicates='drop')

else:

raise ValueError("Invalid binning\_type. Choose 'equal\_width' or 'equal\_frequency'.")

return binned\_data

csv\_file\_path = 'Heart\_disease\_statlog.csv'

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

outcome\_column = 'target'

def calculate\_entropy(data):

"""Calculates the entropy of a pandas Series."""

value\_counts = data.value\_counts(normalize=True)

probabilities = value\_counts.values

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

return entropy

def calculate\_information\_gain(data, feature\_column, target\_column):

"""Calculates the information gain for a given feature."""

total\_entropy = calculate\_entropy(data[target\_column])

weighted\_entropy = 0

for value in data[feature\_column].unique():

subset = data[data[feature\_column] == value]

weighted\_entropy += (len(subset) / len(data)) \* calculate\_entropy(subset[target\_column])

information\_gain = total\_entropy - weighted\_entropy

return information\_gain

def find\_best\_root\_node(data, features, target\_column):

"""Finds the best feature for the root node using Information Gain."""

best\_feature = None

max\_information\_gain = -1 # Initialize with a negative value

# Create a copy of the DataFrame to avoid modifying the original

data\_copy = data.copy()

with warnings.catch\_warnings():

warnings.filterwarnings("ignore", category=DeprecationWarning)

for feature in features:

# Check if the feature is numerical

if np.issubdtype(data\_copy[feature].dtype, np.number):

data\_copy[feature] = data\_copy[feature].astype(np.int64) # Keep as np.int64

information\_gain = calculate\_information\_gain(data\_copy, feature, target\_column)

if information\_gain > max\_information\_gain:

max\_information\_gain = information\_gain

best\_feature = feature

return best\_feature

def build\_decision\_tree(data, features, target\_column):

"""Builds the decision tree recursively."""

# Base case: If all target values are the same, return the target value

if len(data[target\_column].unique()) == 1:

return np.int64(data[target\_column].iloc[0]) # Ensure it's stored as np.int64

# Base case: If no features left, return the most common target value

if len(features) == 0:

return np.int64(data[target\_column].mode()[0])

# Find the best feature to split on

best\_feature = find\_best\_root\_node(data, features, target\_column)

# Create a tree node

tree = {best\_feature: {}}

# Recursively build subtrees for each feature value

for value in sorted(data[best\_feature].unique()): # Sort for consistency

subset = data[data[best\_feature] == value]

remaining\_features = [f for f in features if f != best\_feature]

subtree = build\_decision\_tree(subset, remaining\_features, target\_column)

tree[best\_feature][np.int64(value)] = subtree # Ensure keys are np.int64

return tree

# Example usage:

# Assuming 'df' is your pandas DataFrame, 'features' is a list of feature column names,

# and 'target\_column' is the name of the target variable column.

features = ['sex', 'cp', 'chol'] # Replace with your feature names

target\_column = 'target' # Replace with your target column name

decision\_tree = build\_decision\_tree(df, features, target\_column)

print(decision\_tree)

**Q6.**

import pandas as pd

import numpy as np

import warnings

from sklearn.tree import DecisionTreeClassifier, plot\_tree

import matplotlib.pyplot as plt

# ... (binning, calculate\_entropy, calculate\_information\_gain, find\_best\_root\_node, build\_decision\_tree functions remain the same) ...

# Assuming 'df' is your pandas DataFrame, 'features' is a list of feature column names,

# and 'target\_column' is the name of the target variable column.

features = ['sex', 'cp', 'chol'] # Replace with your feature names

target\_column = 'target' # Replace with your target column name

# Convert continuous features to categorical using binning if needed

for feature in features:

if pd.api.types.is\_numeric\_dtype(df[feature]): # Check if the feature is numeric

df[feature] = binning(df[feature])

# Create a DecisionTreeClassifier object

clf = DecisionTreeClassifier()

# Train the decision tree using your data

clf.fit(df[features], df[target\_column])

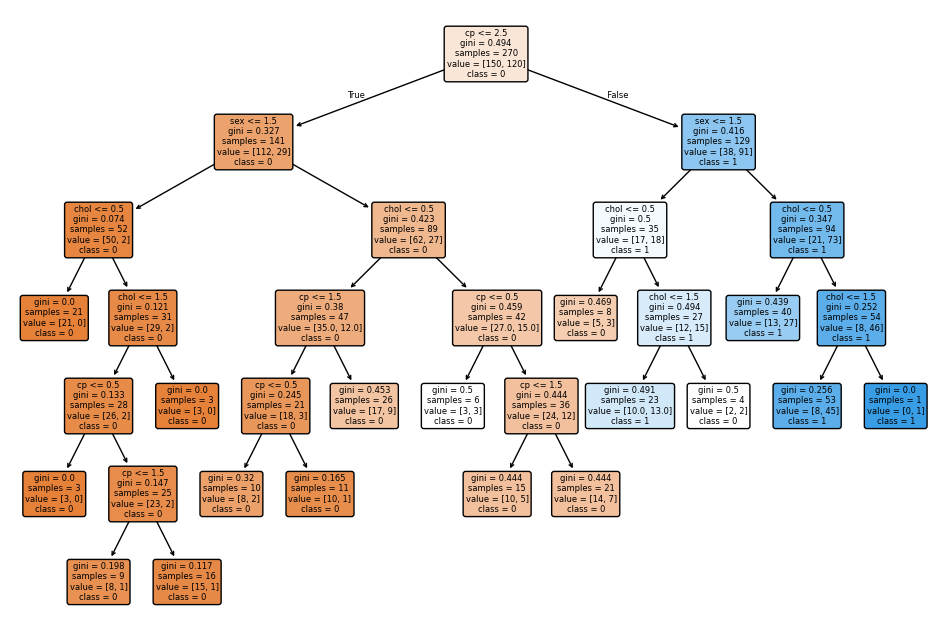
# Plot the decision tree

plt.figure(figsize=(12, 8)) # Adjust figure size as needed

plot\_tree(clf, feature\_names=features, class\_names=['0', '1'], filled=True, rounded=True)

plt.show()

plt.show()



**Q7.**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier

from sklearn.inspection import DecisionBoundaryDisplay

# Assuming 'df' is your pandas DataFrame

# Select two features for visualization (e.g., 'feature1' and 'feature2')

feature1 = 'chol' # Replace with your first feature name

feature2 = 'cp' # Replace with your second feature name

features = [feature1, feature2]

target\_column = 'target' # Replace with your target column name

# Create a DecisionTreeClassifier object

clf = DecisionTreeClassifier()

# Train the decision tree using the selected features and target

clf.fit(df[features], df[target\_column])

# Create a meshgrid for visualization

x\_min, x\_max = df[feature1].min() - 1, df[feature1].max() + 1

y\_min, y\_max = df[feature2].min() - 1, df[feature2].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

# Plot the decision boundary

disp = DecisionBoundaryDisplay.from\_estimator(

clf,

df[features],

response\_method="predict",

xlabel=feature1,

ylabel=feature2,

alpha=0.8,

cmap=plt.cm.RdYlBu,

)

# Plot the training points

disp.ax\_.scatter(

df[feature1], df[feature2], c=df[target\_column], edgecolor="k", cmap=plt.cm.RdYlBu

)

plt.title("Decision Boundary of Decision Tree")

plt.show()

