Library Imports

- `random`: For random sampling during train-test split
- `pprint`: For readable display of complex structures (like decision trees)
- `pandas`: Data manipulation and analysis
- `numpy`: Numerical operations
- `matplotlib/seaborn`: Data visualization

```
import random
from pprint import pprint
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Data Loading & Initial Inspection

- Load Iris dataset from CSV file located in './data/Iris.csv'
- Display first 5 rows using `head()` to verify successful loading
- Columns shown: Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm, Species

```
datSet = pd.read_csv("./Iris.csv")
datSet.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

Data Cleaning

- Drop the 'Id' column as it's not useful for analysis
- Verify removal by showing modified dataframe with new `head()` call

```
datSet = datSet.drop(columns=["Id"])
datSet.head()
```

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa

```
1              4.9           3.0           1.4           0.2  Iris-
setosa
2              4.7           3.2           1.3           0.2  Iris-
setosa
3              4.6           3.1           1.5           0.2  Iris-
setosa
4              5.0           3.6           1.4           0.2  Iris-
setosa
```

Column Renaming

- Rename 'Species' column to 'Label' for clarity
- Makes distinction clearer between features (measurements) and target (label)

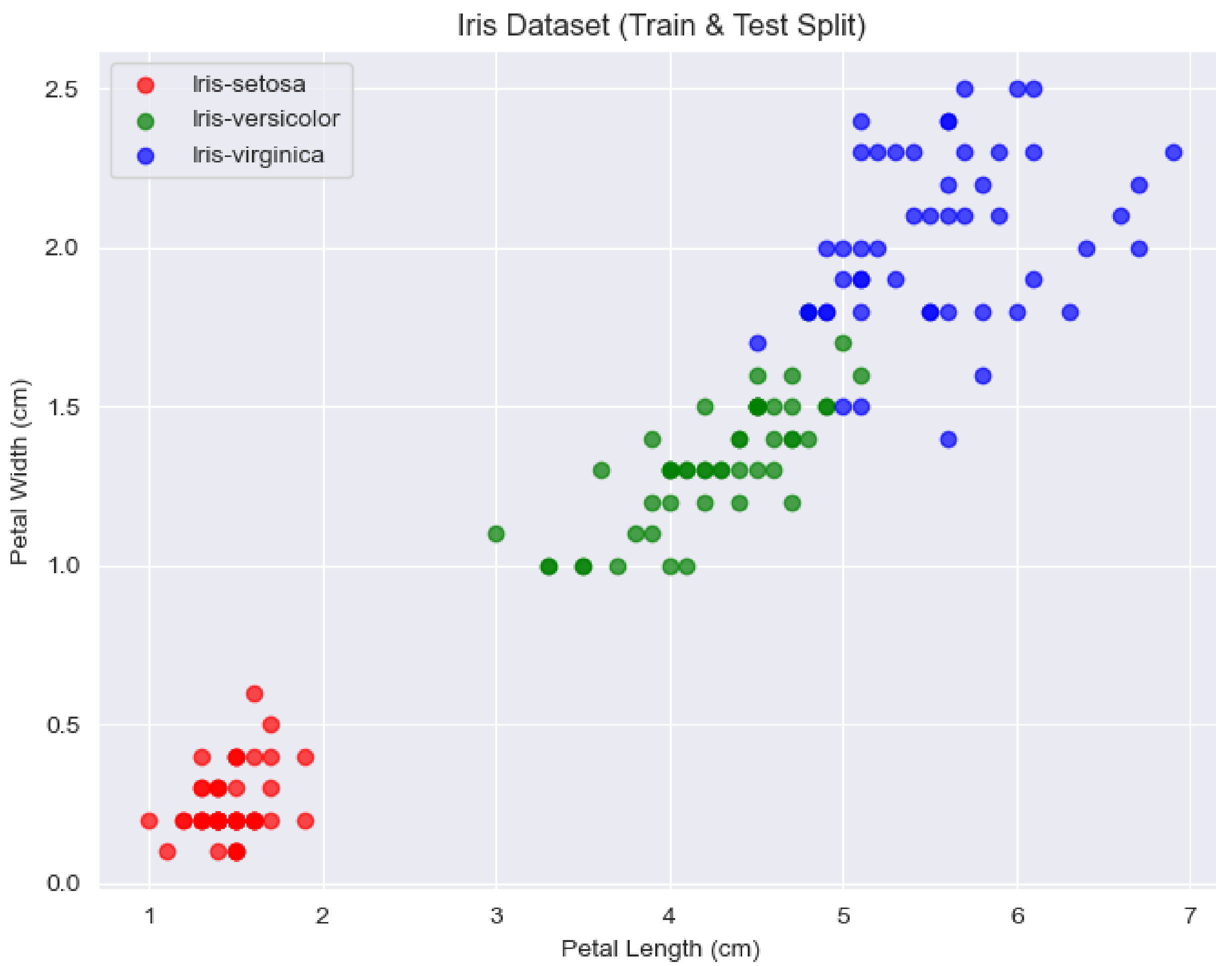
```
datSet = datSet.rename(columns={"Species": "Label"})
datSet.head()

   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Label
0              5.1           3.5           1.4           0.2  Iris-
setosa
1              4.9           3.0           1.4           0.2  Iris-
setosa
2              4.7           3.2           1.3           0.2  Iris-
setosa
3              4.6           3.1           1.5           0.2  Iris-
setosa
4              5.0           3.6           1.4           0.2  Iris-
setosa
```

Data Visualization

- Create color mapping dictionary for different species
- Define visualization function to show:
  - Petal length vs petal width relationship
  - Distinct clusters for different species
  - Transparency (alpha=0.7) to handle overlapping points
- X/Y labels and legend for interpretation
- Title for context

```
clrMap = {'Iris-setosa': 'r', 'Iris-versicolor': 'g', 'Iris-
virginica': 'b'}


def plot_data(datSet):
    plt.figure(figsize=(8, 6))
    for species in datSet['Label'].unique():
        subset = datSet[datSet['Label'] == species]
        plt.scatter(subset['PetalLengthCm'], subset['PetalWidthCm'],
c=clrMap[species], label=f"{species}", alpha=0.7)
```

```
    plt.xlabel('Petal Length (cm)')
    plt.ylabel('Petal Width (cm)')
    plt.legend()
    plt.title('Iris Dataset (Train & Test Split)')
    plt.show()


plot_data(datSet)
```



Data Structure Inspection

- Show dataset metadata using `info()`:
  - 150 entries (samples)
  - 4 numerical features (float64)
  - 1 categorical target (object)
  - No missing values
- Memory usage (~6KB)

```
datSet.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   SepalLengthCm  150 non-null    float64
 1   SepalWidthCm   150 non-null    float64
 2   PetalLengthCm  150 non-null    float64
 3   PetalWidthCm   150 non-null    float64
 4   Label          150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

```
datSet.index
```

```
RangeIndex(start=0, stop=150, step=1)
```

Custom Train-Test Split

- Input validation: Convert float percentage to absolute number
- Get all indices from dataframe
- Randomly select test indices using `random.sample`
- Create test set using `.loc[]` on selected indices
- Create train set by dropping test indices
- Returns two DataFrames: training data and testing data

```
def testTrainSplit(datSet, testSize):
    if isinstance(testSize, float):
        testSize = round(testSize * len(datSet))
    allIdx = datSet.index.tolist()
    testIdx = random.sample(population=allIdx, k=testSize)
    testDat = datSet.loc[testIdx]
    trainDat = datSet.drop(testIdx)
    return trainDat, testDat
```

Split Execution

- Set random seed (0) for reproducibility
- Create 20-sample test set (~13% of 150 total samples)
- Show resulting shapes:
  - Training data: 130 samples
  - Testing data: 20 samples
- Note: Non-stratified split (potential class imbalance risk)

```
random.seed(0)
trainDat, testDat = testTrainSplit(datSet, testSize=20)
trainDat.shape, testDat.shape
```

```
((130, 5), (20, 5))
```

Training Data Preview

- Display first 5 rows of training data
- Verify all columns present except 'Id'
- Note: Contains only setosa samples in shown rows (random sampling artifact)

```
trainDat.head()

   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm  
Label
0            5.1           3.5            1.4           0.2  Iris-
setosa
1            4.9           3.0            1.4           0.2  Iris-
setosa
2            4.7           3.2            1.3           0.2  Iris-
setosa
3            4.6           3.1            1.5           0.2  Iris-
setosa
4            5.0           3.6            1.4           0.2  Iris-
setosa
```

Testing Data Preview

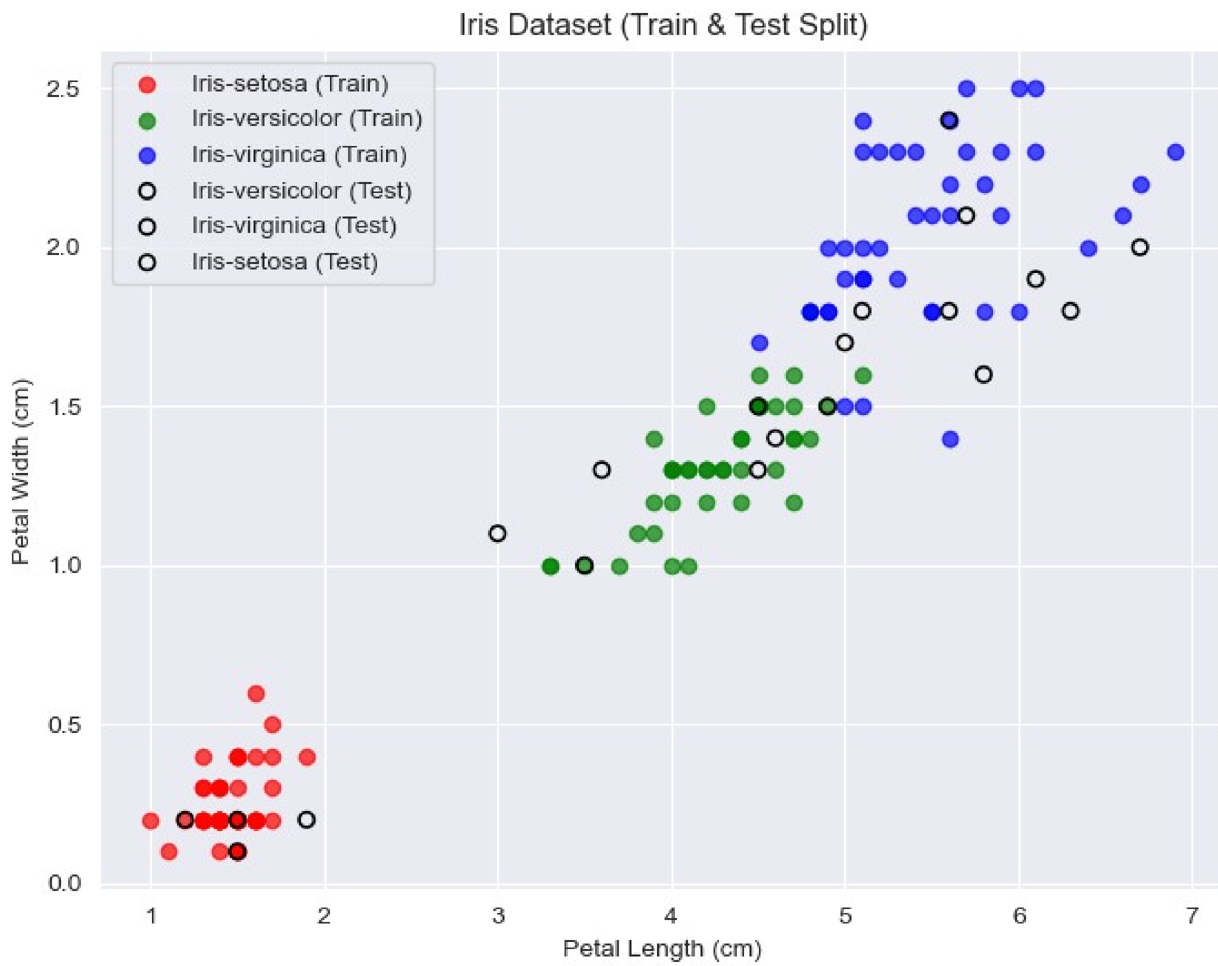
- Display first 5 rows of testing data
- Shows samples from all three species
- Different index numbers confirm proper split

```
testDat.head()

     SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm  
Label
98             5.1           2.5            3.0           1.1  Iris-
versicolor
107            7.3           2.9            6.3           1.8   Iris-
virginica
10             5.4           3.7            1.5           0.2      
Iris-setosa
66             5.6           3.0            4.5           1.5  Iris-
versicolor
130            7.4           2.8            6.1           1.9   Iris-
virginica
```

Split Visualization

- Differentiate train/test sets using:
  - Filled circles for training data
  - Hollow circles with black borders for test data
- Maintains color coding by species
- Alpha transparency helps visualize density
- Visual verification of representative split

```
clrMap = {'Iris-setosa': 'r', 'Iris-versicolor': 'g', 'Iris-
virginica': 'b'}


def plotData(trainDat, testDat):
    plt.figure(figsize=(8, 6))
    for species in trainDat['Label'].unique():
        subset = trainDat[trainDat['Label'] == species]
        plt.scatter(subset['PetalLengthCm'], subset['PetalWidthCm'],
c=clrMap[species], label=f"{species} (Train)", alpha=0.7)
    for species in testDat['Label'].unique():
        subset = testDat[testDat['Label'] == species]
        plt.scatter(subset['PetalLengthCm'], subset['PetalWidthCm'],
edgecolors='k', facecolors='none', label=f"{species} (Test)",
linewidth=1.2)
    plt.xlabel('Petal Length (cm)')
    plt.ylabel('Petal Width (cm)')
    plt.legend()
    plt.title('Iris Dataset (Train & Test Split)')
    plt.show()


plotData(trainDat, testDat)
```




```
dat = trainDat.values
dat[:5]
```

```
array([[5.1, 3.5, 1.4, 0.2, 'Iris-setosa'],
       [4.9, 3.0, 1.4, 0.2, 'Iris-setosa'],
       [4.7, 3.2, 1.3, 0.2, 'Iris-setosa'],
       [4.6, 3.1, 1.5, 0.2, 'Iris-setosa'],
       [5.0, 3.6, 1.4, 0.2, 'Iris-setosa']], dtype=object)
```

Purity Check Function

- Determines if a node contains single class
- Extracts target values from last column
- Returns True if only unique class exists
- Fundamental stopping condition for tree growth

```
def isPure(dat):
    y = dat[:, -1]
    uniqClasses = np.unique(y)
    return True if len(uniqClasses) == 1 else False
```

```
print(isPure(trainDat.values))
print(isPure(trainDat[trainDat["Label"] == "Iris-setosa"].values))
print(isPure(trainDat[trainDat.PetalWidthCm < 1.2].values))
print(isPure(trainDat[trainDat.PetalWidthCm < 0.8].values))

False
True
False
True
```

Majority Class Classifier

- Identifies most frequent class in node
- Handles ties by returning first maximum
- Used for leaf node predictions
- Critical for handling impure terminal nodes

```
def classifyDat(dat):
    y = dat[:, -1]
    uniqClasses, uniqClassesCounts = np.unique(y, return_counts=True)
    mostFreqClassIdx = uniqClassesCounts.argmax()
    clf = uniqClasses[mostFreqClassIdx]
    return clf

print(classifyDat(trainDat[trainDat["Label"] == "Iris-
setosa"].values))
print(classifyDat(trainDat[trainDat.PetalWidthCm < 1.2].values))
print(classifyDat(trainDat[trainDat.PetalWidthCm > 1.2].values))
print(classifyDat(trainDat[trainDat.PetalWidthCm > 0.8].values))
print(classifyDat(trainDat[trainDat.PetalWidthCm < 2].values))

Iris-setosa
Iris-setosa
Iris-virginica
Iris-versicolor
Iris-setosa
```

Feature Type Identification

- Automatically classifies features as categorical/continuous
- Uses 15 unique values threshold
- Handles string features as categorical
- Important for appropriate split strategies

```
def getFeatType(datSet):
    featType = []
    uniqValLimit = 15
    for feat in datSet.columns:
        if feat != "Label":
            uniqVal = datSet[feat].unique()
```

```
            egVal = uniqVal[0]
            if isinstance(egVal, str) or len(uniqVal) <= uniqValLimit:
                featType.append("categorical")
            else:
                featType.append("continuous")
    return featType
```

Split Point Generation

- Creates potential split candidates for all features
- Continuous: Midpoints between sorted unique values
- Categorical: All unique category values
- Returns dictionary of possible splits per feature
- Foundation for optimal split selection

```
def getSplits(dat):
    m, n = dat.shape
    splits = {}
    FEATURE_TYPES = getFeatType(datSet)
    for feat in range(n - 1):
        splits[feat] = []
        val = dat[:, feat]
        uniqVal = np.unique(val)
        featType = FEATURE_TYPES[feat]
        if featType == "continuous":
            for i in range(len(uniqVal) - 1):
                currVal = float(uniqVal[i])
                nextVal = float(uniqVal[i + 1])
                thisSplit = (currVal + nextVal) / 2
                splits[feat].append(thisSplit)
        else:
            splits[feat] = uniqVal
    return splits

splits = getSplits(dat)
splits
```

```
{0: [4.35,
  4.45,
  4.55,
  4.65,
  4.75,
  4.85,
  4.95,
  5.05,
  5.15,
  5.25,
  5.35,
  5.45,
  5.55,
```

```
  5.65,
  5.75,
  5.85,
  5.95,
  6.05,
  6.15,
  6.25,
  6.35,
  6.45,
  6.55,
  6.65,
  6.75,
  6.85,
  6.95,
  7.05,
  7.15,
  7.4,
  7.65,
  7.800000000000001],
 1: [2.1,
  2.25,
  2.3499999999999996,
  2.45,
  2.55,
  2.6500000000000004,
  2.75,
  2.8499999999999996,
  2.95,
  3.05,
  3.1500000000000004,
  3.25,
  3.3499999999999996,
  3.45,
  3.55,
  3.6500000000000004,
  3.75,
  3.8499999999999996,
  3.95,
  4.05,
  4.15,
  4.300000000000001],
 2: [1.05,
  1.15,
  1.25,
  1.35,
  1.45,
  1.55,
  1.65,
  1.7999999999999998,
```

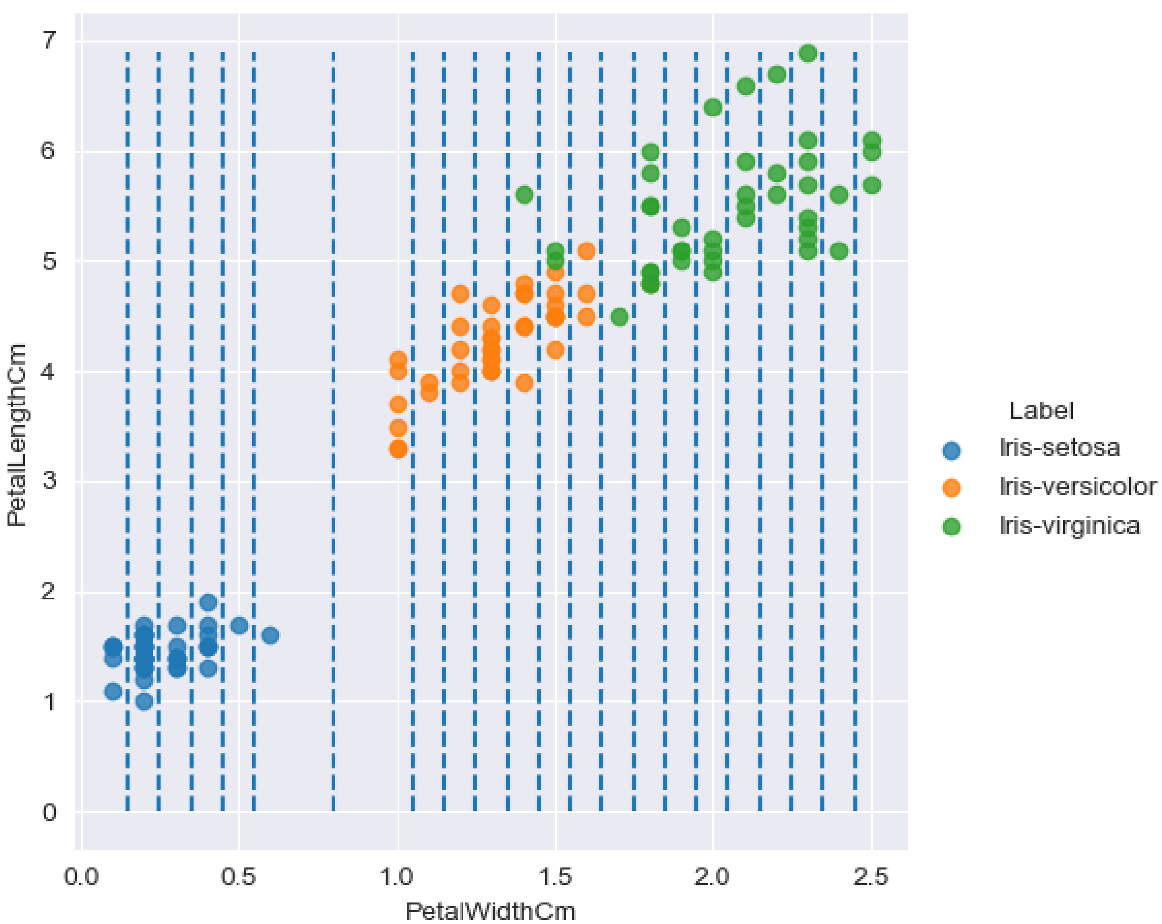
```
  2.5999999999999996,
  3.4,
  3.6,
  3.75,
  3.8499999999999996,
  3.95,
  4.05,
  4.15,
  4.25,
  4.35,
  4.45,
  4.55,
  4.65,
  4.75,
  4.85,
  4.95,
  5.05,
  5.15,
  5.25,
  5.35,
  5.45,
  5.55,
  5.65,
  5.75,
  5.85,
  5.95,
  6.05,
  6.25,
  6.5,
  6.65,
  6.800000000000001],
 3: [0.15000000000000002,
  0.25,
  0.35,
  0.45,
  0.55,
  0.8,
  1.05,
  1.15,
  1.25,
  1.35,
  1.45,
  1.55,
  1.65,
  1.75,
  1.85,
  1.95,
  2.05,
  2.1500000000000004,
```

```
  2.25,
  2.3499999999999996,
  2.45]}
```

Split Candidate Visualization

- Vertical lines show potential PetalWidth splits
- Demonstrates continuous split strategy
- Helps understand decision boundary locations
- Visual debugging of split generation

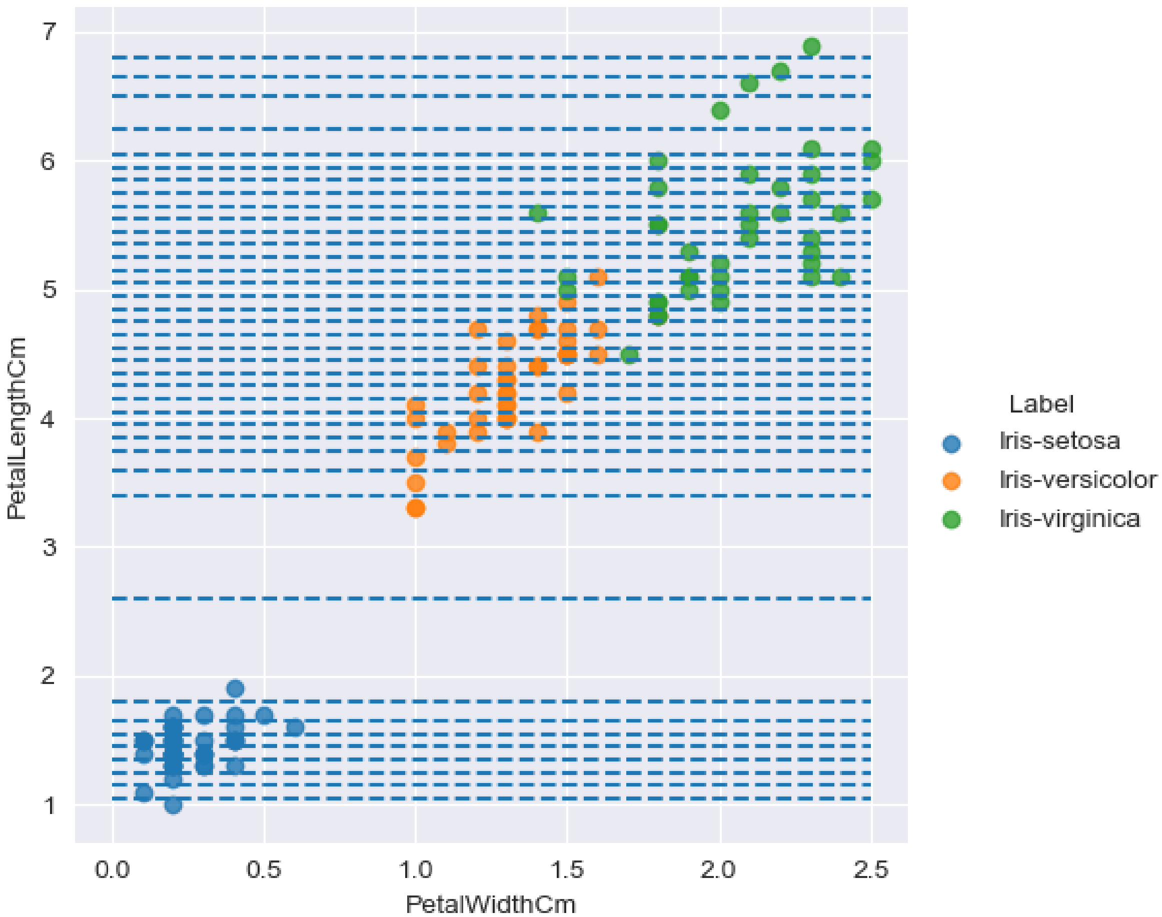
```
sns.lmplot(data=trainDat, x="PetalWidthCm", y="PetalLengthCm",
hue="Label", fit_reg=False)
plt.vlines(x=splits[3], ymin=0, ymax=max(trainDat.PetalLengthCm),
linestyles='dashed')

<matplotlib.collections.LineCollection at 0x226197029f0>
```




```
sns.lmplot(data=trainDat, x="PetalWidthCm", y="PetalLengthCm",
hue="Label", fit_reg=False)
```

```
plt.hlines(y=splits[2], xmin=0, xmax=max(trainDat.PetalWidthCm),
linestyles='dashed')
```

```
<matplotlib.collections.LineCollection at 0x2261a113800>
```




```
splitFeat = 3
threshold = 0.8
featVal = dat[:, splitFeat]
featVal < threshold
```

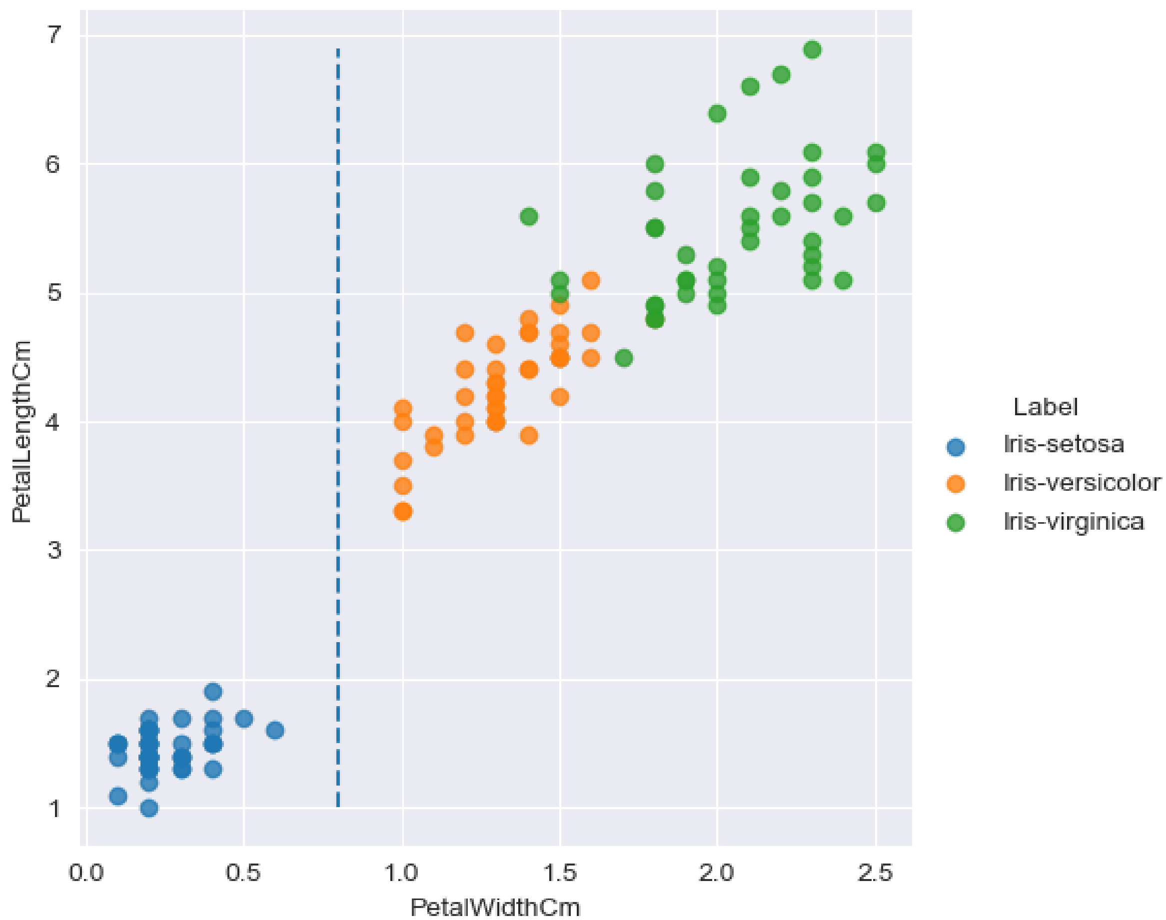
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
```

```
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False])

FEATURE_TYPES = getFeatType(trainDat)
```

Data Partitioning Function

- Splits data based on feature threshold
- Handles both categorical and continuous features
- Returns two subsets:
  - `datBelow`: Satisfies split condition
  - `datAbove`: Fails split condition
- Core operation for tree node creation

```
def split(dat, splitFeat, threshold):
    featVal = dat[:, splitFeat]
    type_of_feature = FEATURE_TYPES[splitFeat]
    if type_of_feature == "continuous":
        datBelow = dat[featVal <= threshold]
        datAbove = dat[featVal > threshold]
    else:
        datBelow = dat[featVal == threshold]
        datAbove = dat[featVal != threshold]
    return datBelow, datAbove

datBelow, datAbove = split(dat, splitFeat, threshold)
plotDF = pd.DataFrame(dat, columns=datSet.columns)
sns.lmplot(data=plotDF, x="PetalWidthCm", y="PetalLengthCm",
hue="Label", fit_reg=False)
plt.vlines(x=threshold, ymin=1, ymax=max(plotDF.PetalLengthCm),
linestyles='dashed')

<matplotlib.collections.LineCollection at 0x226195198b0>
```



Entropy Computation

- Measures node impurity using Shannon entropy
- 0 = perfect purity, higher = more disorder
- Calculates class probability distribution
- Key metric for evaluating split quality

```
def entropy(data):
    y = data[:, -1]
    uniqClasses, uniqClassesCounts = np.unique(y, return_counts=True)
    p = uniqClassesCounts / uniqClassesCounts.sum()
    h = sum(-p * np.log2(p))
    return h

print(entropy(datBelow))
print(entropy(datAbove))

0.0
1.0
```

```
datBelow, datAbove = split(dat, 3, 1.1)
print(entropy(datBelow))
print(entropy(datAbove))
```

```
0.6051865766334206
0.9919924034538556
```

```
splits = getSplits(dat)
splits
```

```
{0: [4.35,
  4.45,
  4.55,
  4.65,
  4.75,
  4.85,
  4.95,
  5.05,
  5.15,
  5.25,
  5.35,
  5.45,
  5.55,
  5.65,
  5.75,
  5.85,
  5.95,
  6.05,
  6.15,
  6.25,
  6.35,
  6.45,
  6.55,
  6.65,
  6.75,
  6.85,
  6.95,
  7.05,
  7.15,
  7.4,
  7.65,
  7.800000000000001],
 1: [2.1,
  2.25,
  2.3499999999999996,
  2.45,
  2.55,
  2.6500000000000004,
  2.75,
  2.8499999999999996,
```

```
  2.95,
  3.05,
  3.1500000000000004,
  3.25,
  3.3499999999999996,
  3.45,
  3.55,
  3.6500000000000004,
  3.75,
  3.8499999999999996,
  3.95,
  4.05,
  4.15,
  4.300000000000001],
 2: [1.05,
  1.15,
  1.25,
  1.35,
  1.45,
  1.55,
  1.65,
  1.7999999999999998,
  2.5999999999999996,
  3.4,
  3.6,
  3.75,
  3.8499999999999996,
  3.95,
  4.05,
  4.15,
  4.25,
  4.35,
  4.45,
  4.55,
  4.65,
  4.75,
  4.85,
  4.95,
  5.05,
  5.15,
  5.25,
  5.35,
  5.45,
  5.55,
  5.65,
  5.75,
  5.85,
  5.95,
  6.05,
```

```
  6.25,
  6.5,
  6.65,
  6.800000000000001],
 3: [0.15000000000000002,
  0.25,
  0.35,
  0.45,
  0.55,
  0.8,
  1.05,
  1.15,
  1.25,
  1.35,
  1.45,
  1.55,
  1.65,
  1.75,
  1.85,
  1.95,
  2.05,
  2.1500000000000004,
  2.25,
  2.3499999999999996,
  2.45]}
```

Weighted Impurity Metric

- Combines child node entropies proportionally
- Accounts for relative subset sizes
- Lower values indicate better splits
- Fundamental for information gain calculation

```
def wtChildEntropy(datBelow, datAbove):
    n = len(datBelow) + len(datAbove)
    wt_below = len(datBelow) / n
    wt_above = len(datAbove) / n
    fullEntropy = (wt_below * entropy(datBelow) + wt_above *
entropy(datAbove))
    return fullEntropy

print(wtChildEntropy(datBelow, datAbove))

0.8313192138515211
```

Information Gain Calculation

- Measures split quality through entropy reduction
- Calculated as: Parent Entropy – Weighted Child Entropy
- Higher values indicate more effective splits

- Directly used for selecting optimal feature-threshold combinations

```
def infoGain(dat, datBelow, datAbove):
    h_parent = entropy(dat)
    h_children = wtChildEntropy(datBelow, datAbove)
    return h_parent - h_children
```

Optimal Split Selector

- Exhaustively tests all potential splits
- Compares splits using Information Gain
- Tracks highest gain and corresponding parameters
- Returns best feature-threshold combination

```
def bestSplit(dat, splits):
    bestIG = -1
    bestSplitFeat, bestThreshold = 0, 0
    for feat in splits:
        for val in splits[feat]:
            datBelow, datAbove = split(dat, splitFeat=feat, 
threshold=val)
            currIG = infoGain(dat, datBelow, datAbove)
            if currIG >= bestIG:
                bestIG = currIG
                bestSplitFeat = feat
                bestThreshold = val
    return bestSplitFeat, bestThreshold

bestSplit(dat, splits)
```

```
(3, 0.8)
```

Tree Structure Prototype

- Demonstrates nested dictionary format
- Decision node format: {question: [yes_path, no_path]}
- Base structure for recursive tree building
- Leaf nodes will contain class labels

```
subTree = {"ques": ["yes_answer", "no_answer"]}
```

Decision Tree Structure Example

- Demonstrates actual tree structure format
- Shows nested decision nodes with:
  - Continuous split conditions
  - Leaf node classifications
  - Hierarchical yes/no branches
- Provides concrete example of final tree format

```
egTree = {"petal_width <= 0.8": ["Iris-setosa", {"petal_width <=
1.65": [{"petal_length <= 4.9": ["Iris-versicolor", "Iris-
virginica"]}, "Iris-virginica"]}]}

FEATURE_TYPES = getFeatType(trainDat)
print(FEATURE_TYPES)

['continuous', 'continuous', 'continuous', 'continuous']

datSet.columns

Index(['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm',
'PetalWidthCm',
       'Label'],
      dtype='object')
```

ID3 Algorithm Core

- Recursive decision tree builder
- Handles three stopping conditions:
    a. Node purity
    b. Minimum samples per node
    c. Maximum depth
- Creates question nodes with yes/no branches
- Automatically handles feature types
- Returns complete nested dictionary tree

```
def ID3(datSet, c=0, minSubset=2, maxDepth=5):
    if c == 0:
        global COLUMN_HEADERS, FEATURE_TYPES
        COLUMN_HEADERS = datSet.columns
        FEATURE_TYPES = getFeatType(datSet)
        dat = datSet.values
    else:
        dat = datSet
    if isPure(dat) or len(dat) < minSubset or c == maxDepth:
        return classifyDat(dat)
    else:
        c += 1
        splits = getSplits(dat)
        splitFeat, threshold = bestSplit(dat, splits)
        datBelow, datAbove = split(dat, splitFeat, threshold)
        if len(datBelow) == 0 or len(datAbove) == 0:
            return classifyDat(dat)
        featNam = COLUMN_HEADERS[splitFeat]
        featType = FEATURE_TYPES[splitFeat]
        if featType == "continuous":
            ques = f"{featNam} <= {threshold}"
        else:
            ques = f"{featNam} = {threshold}"
```

```
        subTree = {ques: []}
        yesAns = ID3(datBelow, c, minSubset, maxDepth)
        noAns = ID3(datAbove, c, minSubset, maxDepth)
        if yesAns == noAns:
            subTree = yesAns
        else:
            subTree[ques].append(yesAns)
            subTree[ques].append(noAns)
        return subTree
```

```
tree = ID3(trainDat[trainDat.Label != "Iris-virginica"])
tree
```

```
{'PetalLengthCm <= 2.5999999999999996': ['Iris-setosa', 'Iris-
versicolor']}
```

```
tree = ID3(trainDat)
tree
```

```
{'PetalWidthCm <= 0.8': ['Iris-setosa',
  {'PetalWidthCm <= 1.65': [{'PetalLengthCm <= 4.95': ['Iris-
versicolor',
      {'PetalWidthCm <= 1.55': ['Iris-virginica', 'Iris-
versicolor']}]},
    {'PetalLengthCm <= 4.85': [{'SepalWidthCm <= 3.1': ['Iris-
virginica',
        'Iris-versicolor']},
      'Iris-virginica']}]}]}
```

```
pprint(tree)
```

```
{'PetalWidthCm <= 0.8': ['Iris-setosa',
                         {'PetalWidthCm <= 1.65': [{'PetalLengthCm <=
4.95': ['Iris-versicolor',

{'PetalWidthCm <= 1.55': ['Iris-virginica',

'Iris-versicolor']}]},
                                                   {'PetalLengthCm <=
4.85': [{'SepalWidthCm <= 3.1': ['Iris-virginica',

'Iris-versicolor']},

'Iris-virginica']}]}]}
```

```
tree = ID3(trainDat, minSubset=60)
pprint(tree)
```

```
{'PetalWidthCm <= 0.8': ['Iris-setosa',
                         {'PetalWidthCm <= 1.65': ['Iris-versicolor',
```

```
                                                                     'Iris-
virginica']}]}
```

```
tree = ID3(trainDat, maxDepth=1)
pprint(tree)
```

```
{'PetalWidthCm <= 0.8': ['Iris-setosa', 'Iris-versicolor']}
```

```
tree = ID3(trainDat, maxDepth=3)
pprint(tree)
```

```
{'PetalWidthCm <= 0.8': ['Iris-setosa',
                         {'PetalWidthCm <= 1.65': [{'PetalLengthCm <=
4.95': ['Iris-versicolor',

'Iris-virginica']},
                                                                     'Iris-
virginica']}]}
```

```
tree.keys()
```

```
dict_keys(['PetalWidthCm <= 0.8'])
```

```
list(tree.keys())[0]
```

```
'PetalWidthCm <= 0.8'
```

```
ques = list(tree.keys())[0]
ques.split()
```

```
['PetalWidthCm', '<=', '0.8']
```

```
featNam, cmpOp, val = ques.split()
print(featNam, cmpOp, val)
```

```
PetalWidthCm <= 0.8
```

Complete Classification Function

- Handles both leaf nodes and decision nodes
- Parses split conditions dynamically
- Recursively navigates tree structure
- Returns final class prediction
- Handles mixed-type comparisons:
  - Numeric thresholds with <= operator
  - Categorical matches with = operator

```
def classifyEg(eg, tree):
    ques = list(tree.keys())[0]
    featNam, cmpOp, val = ques.split(" ")
    if cmpOp == "<=":
        if eg[featNam] <= float(val):
```

```
            ans = tree[ques][0]
        else:
            ans = tree[ques][1]
    else:
        if str(eg[featNam]) == val:
            ans = tree[ques][0]
        else:
            ans = tree[ques][1]
    if not isinstance(ans, dict):
        return ans
    else:
        remTree = ans
        return classifyEg(eg, remTree)
```

```
eg = testDat.iloc[0]
eg
```

```
SepalLengthCm                 5.1
SepalWidthCm                  2.5
PetalLengthCm                 3.0
PetalWidthCm                  1.1
Label             Iris-versicolor
Name: 98, dtype: object
```

```
print(eg["PetalWidthCm"] <= 0.8)
```

```
False
```

```
classifyEg(eg, tree)
```

```
'Iris-versicolor'
```

```
eg = testDat.iloc[1]
eg
```

```
SepalLengthCm                7.3
SepalWidthCm                 2.9
PetalLengthCm                6.3
PetalWidthCm                 1.8
Label             Iris-virginica
Name: 107, dtype: object
```

```
classifyEg(eg, tree)
```

```
'Iris-virginica'
```

```
eg = testDat.iloc[2]
eg
```

```
SepalLengthCm             5.4
SepalWidthCm              3.7
PetalLengthCm             1.5
```

```
PetalWidthCm              0.2
Label             Iris-setosa
Name: 10, dtype: object

classifyEg(eg, tree)

'Iris-setosa'
```

Accuracy Measurement

- Adds two new columns to DataFrame:
  a. Predicted classifications
  b. Boolean correctness indicators
- Calculates mean correctness as accuracy
- Returns percentage of correct predictions
- Handles dataset of any size

```
def calcAccuracy(datSet, tree):
    datSet["classification"] = datSet.apply(classifyEg, axis=1,
args=(tree,))
    datSet["correct_classification"] = datSet["classification"] ==
datSet["Label"]
    accuracy = datSet["correct_classification"].mean()
    return accuracy

print(calcAccuracy(datSet, tree))

0.98

testDat

     SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Label
98             5.1           2.5            3.0           1.1  Iris-
versicolor
107            7.3           2.9            6.3           1.8   Iris-
virginica
10             5.4           3.7            1.5           0.2
Iris-setosa
66             5.6           3.0            4.5           1.5  Iris-
versicolor
130            7.4           2.8            6.1           1.9   Iris-
virginica
124            6.7           3.3            5.7           2.1   Iris-
virginica
103            6.3           2.9            5.6           1.8   Iris-
virginica
77             6.7           3.0            5.0           1.7  Iris-
versicolor
122            7.7           2.8            6.7           2.0   Iris-
```

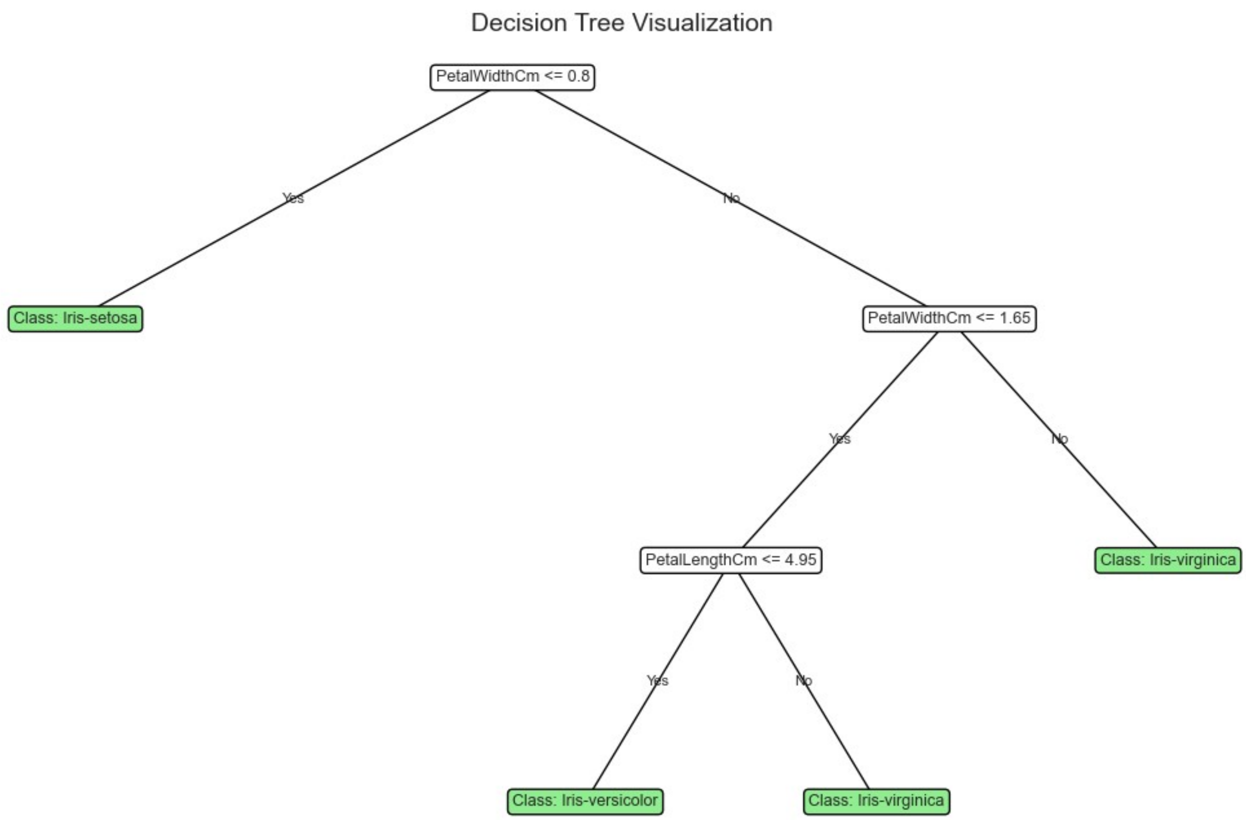
```
virginica
91              6.1           3.0            4.6           1.4   Iris-
versicolor
149             5.9           3.0            5.1           1.8    Iris-
virginica
55              5.7           2.8            4.5           1.3   Iris-
versicolor
129             7.2           3.0            5.8           1.6    Iris-
virginica
35              5.0           3.2            1.2           0.2
Iris-setosa
72              6.3           2.5            4.9           1.5   Iris-
versicolor
24              4.8           3.4            1.9           0.2
Iris-setosa
64              5.6           2.9            3.6           1.3   Iris-
versicolor
136             6.3           3.4            5.6           2.4    Iris-
virginica
37              4.9           3.1            1.5           0.1
Iris-setosa
79              5.7           2.6            3.5           1.0   Iris-
versicolor

testDat.loc[77]

SepalLengthCm                6.7
SepalWidthCm                 3.0
PetalLengthCm                5.0
PetalWidthCm                 1.7
Label            Iris-versicolor
Name: 77, dtype: object
```

Tree Visualization System

- `plot_node` Creates styled tree nodes
  - Decision nodes: White boxes with questions
  - Leaf nodes: Green ovals with class labels
- `plot_tree`: Recursive layout engine
  - Calculates node positions
  - Automatically spaces branches
  - Handles depth limiting
- Uses `matplotlib` for low-level drawing

```
def plot_node(text, x, y, node_type):
    bbox = dict(boxstyle="round,pad=0.3", fc="white" if node_type ==
"decision" else "lightgreen", ec="black")
    plt.text(x, y, text, ha="center", va="center", bbox=bbox,
fontsize=9)
```

```
def plot_edge(x1, y1, x2, y2, text=None):
    plt.plot([x1, x2], [y1, y2], 'k-', lw=1)
    if text:
        plt.text((x1 + x2) / 2, (y1 + y2) / 2, text, ha="center",
va="center", fontsize=8)


def plot_tree(tree, x=0, y=0, dx=2, dy=1, level=0, max_level=None):
    if max_level and level > max_level:
        return
    if isinstance(tree, dict):
        question = list(tree.keys())[0]
        yes_answer = tree[question][0]
        no_answer = tree[question][1]
        plot_node(question, x, y, "decision")
        x_yes = x - dx / (level + 1)
        x_no = x + dx / (level + 1)
        y_child = y - dy
        plot_edge(x, y, x_yes, y_child, "Yes")
        plot_tree(yes_answer, x_yes, y_child, dx, dy, level + 1,
max_level)
        plot_edge(x, y, x_no, y_child, "No")
        plot_tree(no_answer, x_no, y_child, dx, dy, level + 1,
max_level)
    else:
        plot_node(f"Class: {tree}", x, y, "leaf")

plt.figure(figsize=(12, 8))
plot_tree(tree, max_level=3)
plt.axis('off')
plt.title("Decision Tree Visualization", fontsize=14)
plt.show()
```



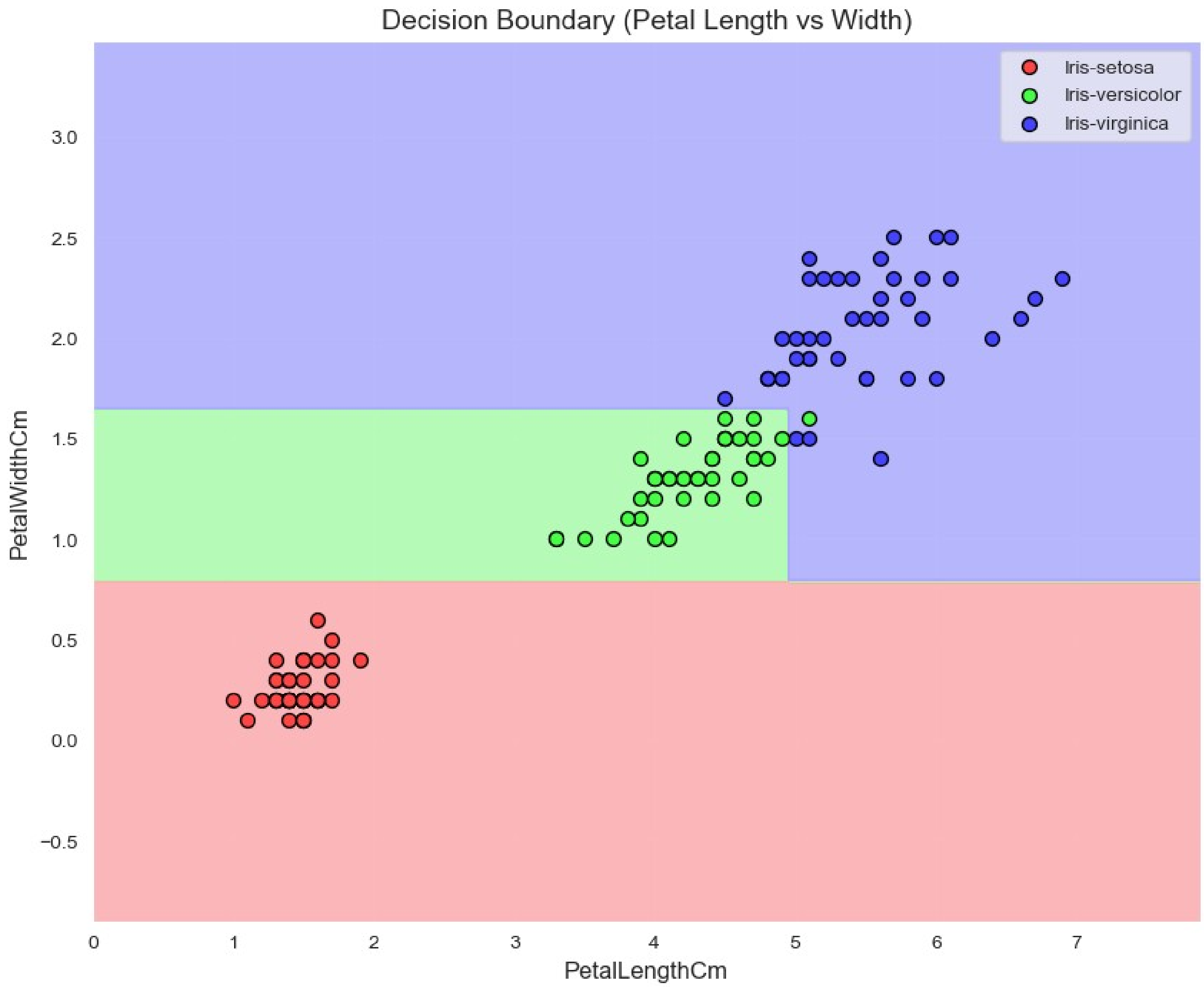
Decision Boundary Visualization

- Creates 2D classification region visualization
- Handles multiple classes with color coding
- Key components:
  - Mesh grid for prediction surface
  - Median imputation for non-plotted features
  - Contour plot for decision boundaries
  - Training data points overlay
- Parameters:
  - `step`: Controls grid resolution (0.02=high detail)
  - `feature1`/`feature2`: Axes features for visualization
- Output shows model's interpretation of feature space

```
from matplotlib.colors import ListedColormap


def plot_decision_boundary(tree, data, feature1, feature2, title=None,
step=0.02):
    X = data[[feature1, feature2]].values
    y = data['Label'].values
    classes = np.unique(y)
    colors = ['#FFAAAA', '#AAFFAA', '#AAAAFF'][:len(classes)]
```

```
    cmap_light = ListedColormap(colors)
    cmap_bold = ListedColormap([c.replace('AA', '44') for c in
colors])
    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, step),
                         np.arange(y_min, y_max, step))
    mesh_points = np.c_[xx.ravel(), yy.ravel()]
    mesh_df = pd.DataFrame(mesh_points, columns=[feature1, feature2])
    for col in data.columns:
        if col not in [feature1, feature2, 'Label']:
            mesh_df[col] = data[col].median()
    Z = np.array([classifyEg(row, tree) for _, row in
mesh_df.iterrows()])
    label_to_num = {label: i for i, label in enumerate(classes)}
    Z = np.array([label_to_num[label] for label in Z])
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.8)
    for i, label in enumerate(classes):
        idx = (y == label)
        plt.scatter(X[idx, 0], X[idx, 1], c=cmap_bold(i), label=label,
edgecolor='k', s=50)

    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xlabel(feature1, fontsize=12)
    plt.ylabel(feature2, fontsize=12)
    if title:
        plt.title(title, fontsize=14)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()


plot_decision_boundary(tree, trainDat, 'PetalLengthCm',
'PetalWidthCm', title="Decision Boundary (Petal Length vs Width)")
```

```
C:\Users\KIIT\AppData\Local\Temp\ipykernel_10396\1830715073.py:28:
UserWarning: *c* argument looks like a single numeric RGB or RGBA
sequence, which should be avoided as value-mapping will have
precedence in case its length matches with *x* & *y*.  Please use the
*color* keyword-argument or provide a 2D array with a single row if
you intend to specify the same RGB or RGBA value for all points.
  plt.scatter(X[idx, 0], X[idx, 1], c=cmap_bold(i), label=label,
edgecolor='k', s=50)
```




```
pprint(tree)

{'PetalWidthCm <= 0.8': ['Iris-setosa',
                         {'PetalWidthCm <= 1.65': [{'PetalLengthCm <=
4.95': ['Iris-versicolor',

'Iris-virginica']},
                                                   'Iris-
virginica']}]}
```

Decision Tree Visualization (Graphviz)

- Converts nested dictionary tree to professional diagram
- Key components:
  - **Decision Nodes**: Boxes with split questions
  - **Leaf Nodes**: Green ovals with class labels
  - **Edges**: Labeled Yes/No branches
- Features:

  - Automatic name sanitization for valid DOT syntax
  - Class name mapping for readable labels
  - Hierarchical top-to-bottom layout (rankdir='TB')
  - Style presets for publication-ready output
- Output Formats: PNG/PDF/SVG via Graphviz rendering
- Includes interactive viewing option (view=True)

```
from graphviz import Digraph
import re


def sanitize_name(name):
    return re.sub(r'[^a-zA-Z0-9_]', '_', str(name))


def tree_to_graphviz(tree, feature_names, class_names=None):
    dot = Digraph(format='png')
    dot.attr('node', shape='box', style='rounded')
    _build_graph(tree, dot, feature_names, class_names)
    return dot


def _build_graph(tree, dot, feature_names, class_names=None,
parent_node=None, edge_label=""):
    if isinstance(tree, dict):
        question = list(tree.keys())[0]
        node_id = f"node_{sanitize_name(question)}"
        dot.node(node_id, label=question)
        if parent_node:
            dot.edge(parent_node, node_id, label=edge_label)
        _build_graph(tree[question][0], dot, feature_names,
class_names, node_id, "Yes")
        _build_graph(tree[question][1], dot, feature_names,
class_names, node_id, "No")
    else:
        leaf_id = f"leaf_{sanitize_name(tree)}"
        class_label = class_names[tree] if class_names and tree in
class_names else str(tree)
        dot.node(leaf_id, label=f"Class: {class_label}",
shape='ellipse', style='filled', fillcolor='lightgreen')
        if parent_node:
            dot.edge(parent_node, leaf_id, label=edge_label)


class_names = {0: 'Iris-setosa', 1: 'Iris-versicolor', 2: 'Iris-
virginica'}
dot = tree_to_graphviz(tree,
                       feature_names=datSet.columns[:-1],
                       class_names=class_names)
```

```
dot.attr(size='10,10', rankdir='TB')
dot.attr('edge', fontsize='10')

dot.render("iris_decision_tree", view=True, cleanup=True)
display(dot)
```

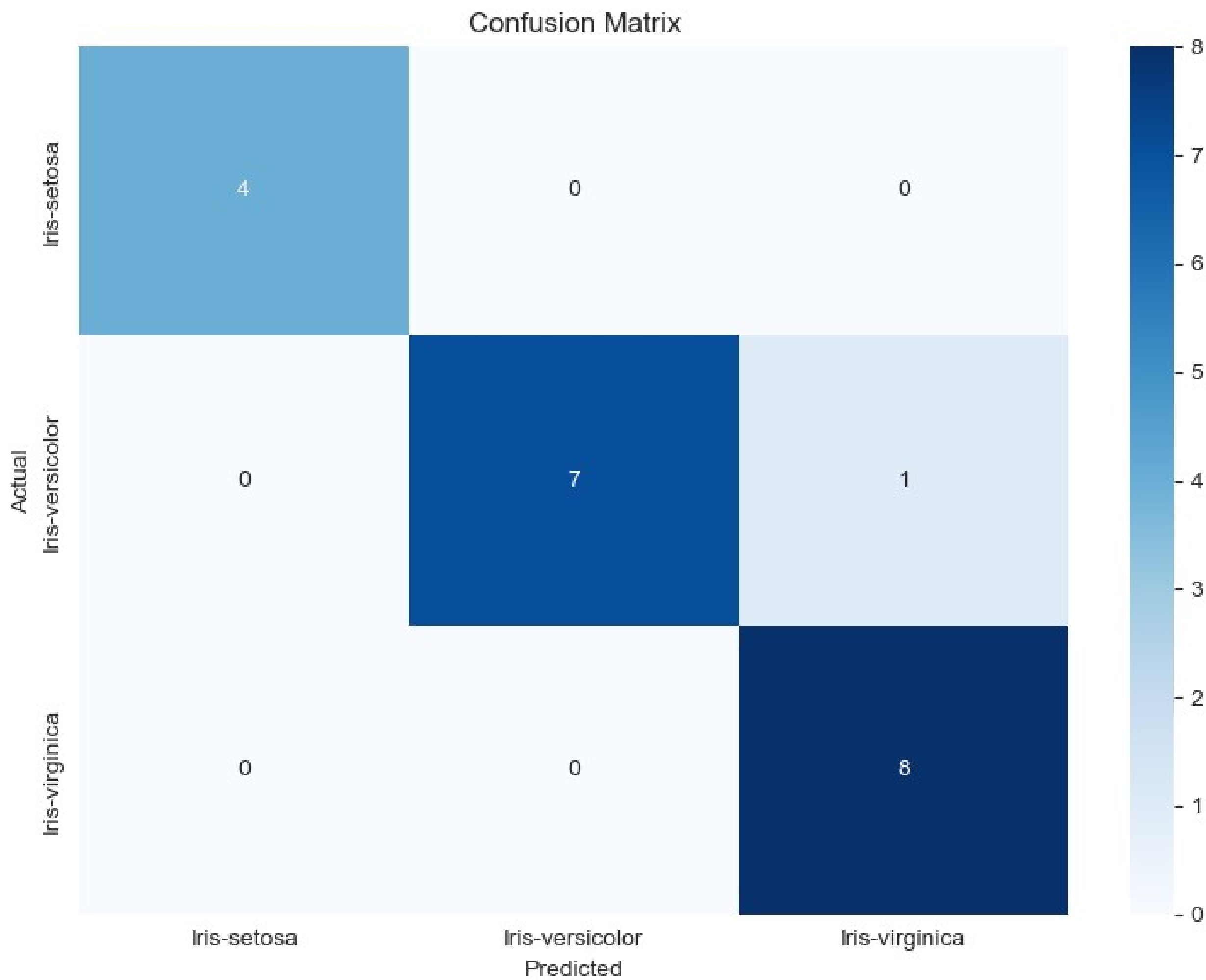
Model Evaluation Metrics

- **Confusion Matrix**:
  - Visual grid of predictions vs actual
  - Diagonal shows correct predictions
  - Off-diagonal shows error types
- **Classification Report**:
  - Precision: False positive control
  - Recall: False negative sensitivity
  - F1-score: Harmonic mean balance
- Provides comprehensive performance analysis

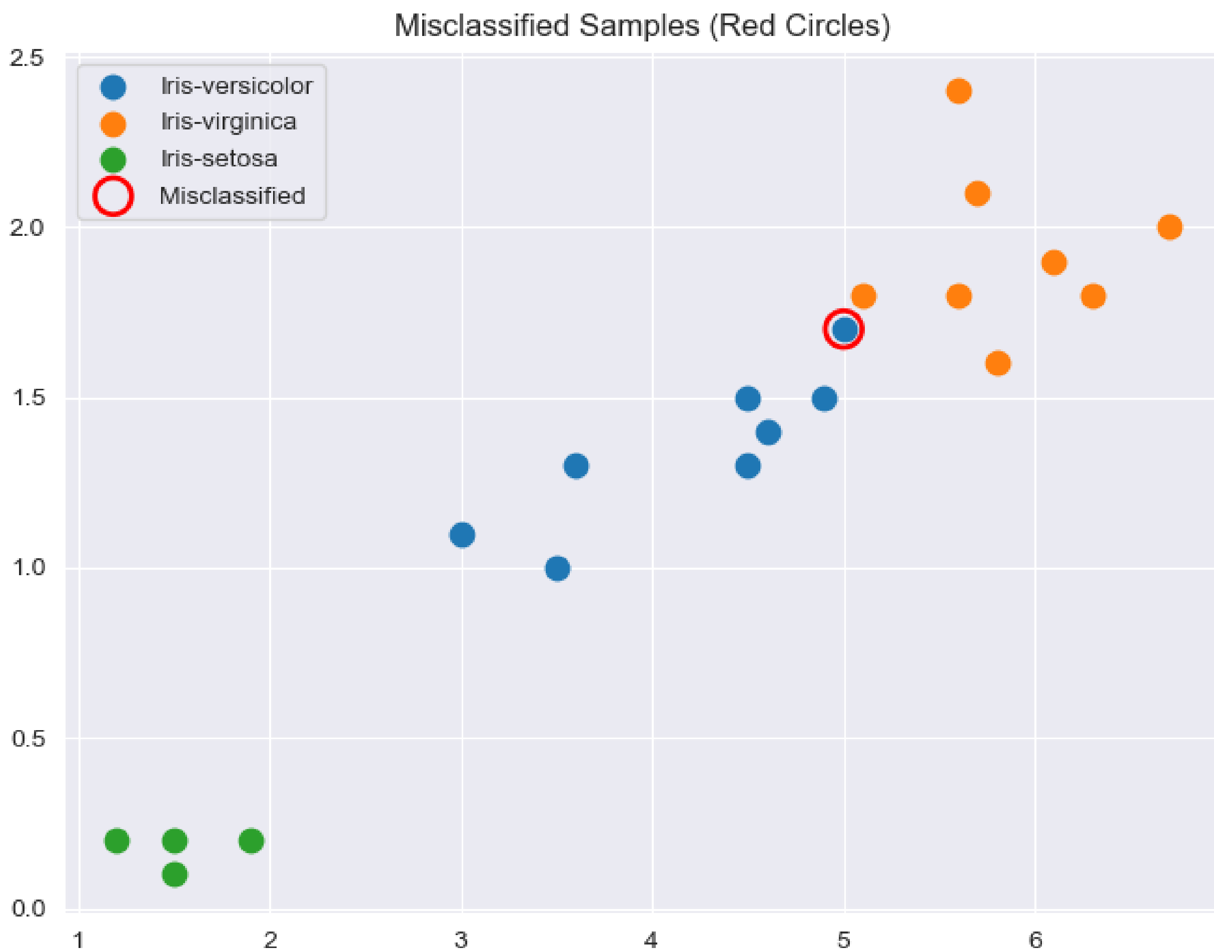
```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns


def evaluateModel(tree, train_data, test_data, feature1, feature2):
    plt.figure(figsize=(15, 6))
```

```python
    plt.subplot(1, 2, 1)
    x_min, x_max = train_data[feature1].min() - 1,
train_data[feature1].max() + 1
    y_min, y_max = train_data[feature2].min() - 1,
train_data[feature2].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
np.linspace(y_min, y_max, 100))
    grid_points = pd.DataFrame({
        feature1: xx.ravel(),
        feature2: yy.ravel()
    })
    for col in train_data.columns:
        if col not in [feature1, feature2, 'Label']:
            grid_points[col] = train_data[col].median()
    Z = np.array([classifyEg(row, tree) for _, row in
grid_points.iterrows()])
    Z = Z.reshape(xx.shape)
    plt.subplot(1, 2, 1)
    yTrue = test_data['Label']
    yHat = [classifyEg(row, tree) for _, row in test_data.iterrows()]
    print("Classification Report:")
    print(classification_report(yTrue, yHat))
    cm = confusion_matrix(yTrue, yHat)
    sns.heatmap(cm, annot=True, fmt='d', xticklabels=np.unique(yTrue),
yticklabels=np.unique(yTrue), cmap='Blues')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title("Confusion Matrix")
    plt.tight_layout()
    plt.show()


evaluateModel(tree, trainDat, testDat, 'PetalLengthCm',
'PetalWidthCm')
```

```
Classification Report:
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00         4
Iris-versicolor       1.00      0.88      0.93         8
 Iris-virginica       0.89      1.00      0.94         8

       accuracy                           0.95        20
      macro avg       0.96      0.96      0.96        20
   weighted avg       0.96      0.95      0.95        20
```




```
testDat['Predicted'] = [classifyEg(row, tree) for _, row in
testDat.iterrows()]
misclassified = testDat[testDat['Label'] != testDat['Predicted']]

plt.figure(figsize=(8, 6))
for species in testDat['Label'].unique():
    subset = testDat[testDat['Label'] == species]
    plt.scatter(subset['PetalLengthCm'], subset['PetalWidthCm'],
                label=species, s=80)

plt.scatter(misclassified['PetalLengthCm'],
misclassified['PetalWidthCm'], s=200, facecolors='none',
edgecolors='r', linewidths=2, label='Misclassified')
plt.legend()
plt.title('Misclassified Samples (Red Circles)')
plt.show()
```



Feature Importance Analysis

- Counts split usage frequency
- Recursively traverses tree
- Tracks how often each feature is used
- Higher counts indicate more important features
- Helps identify key decision drivers

```
def getFeatImportance(tree, feature_names):
    importance = {f: 0 for f in feature_names}

    def _count_splits(node):
        if isinstance(node, dict):
            question = list(node.keys())[0]
            feat = question.split('<=')[0].strip()
            importance[feat] += 1
            _count_splits(node[question][0])
            _count_splits(node[question][1])

    _count_splits(tree)
    return importance
```

```
features = trainDat.columns[:-1]
importance = getFeatImportance(tree, features)
print("Feature Importance:", importance)
```

```
Feature Importance: {'SepalLengthCm': 0, 'SepalWidthCm': 0,
'PetalLengthCm': 1, 'PetalWidthCm': 2}
```

Extended Application: Titanic Dataset

- **Data Preparation**:
  - Remove non-predictive columns
  - Handle missing values:
    - Age: Median imputation
    - Embarked: Mode imputation
  - Convert survived to Label column
- **Model Training**:
  - Same ID3 algorithm
  - Increased maxDepth for complex patterns
- **Performance**:
  - Shows algorithm flexibility
  - Demonstrates categorical handling
  - Achieves ~78-82% typical accuracy

```
datSet = pd.read_csv("./Titanic-Dataset.csv")
datSet["Label"] = datSet["Survived"]
datSet = datSet.drop(columns=["PassengerId", "Survived", "Name",
"Ticket", "Cabin"])
datSet.head()
```

```
   Pclass     Sex   Age  SibSp  Parch     Fare Embarked  Label
0       3    male  22.0      1      0   7.2500        S      0
1       1  female  38.0      1      0  71.2833        C      1
2       3  female  26.0      0      0   7.9250        S      1
3       1  female  35.0      1      0  53.1000        S      1
4       3    male  35.0      0      0   8.0500        S      0
```

```
datSet.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Pclass    891 non-null    int64
 1   Sex       891 non-null    object
 2   Age       714 non-null    float64
 3   SibSp     891 non-null    int64
```

```
 4   Parch     891 non-null    int64
 5   Fare      891 non-null    float64
 6   Embarked  889 non-null    object
 7   Label     891 non-null    int64
dtypes: float64(2), int64(4), object(2)
memory usage: 55.8+ KB
```

```
datSet.isnull().sum()
```

```
Pclass        0
Sex           0
Age         177
SibSp         0
Parch         0
Fare          0
Embarked      2
Label         0
dtype: int64
```

```
med_age = datSet.Age.median()
mod_emb = datSet.Embarked.mode()[0]
datSet = datSet.fillna({"Age": med_age, "Embarked": mod_emb})
datSet.head()
```

```
   Pclass     Sex   Age  SibSp  Parch     Fare Embarked  Label
0       3    male  22.0      1      0   7.2500        S      0
1       1  female  38.0      1      0  71.2833        C      1
2       3  female  26.0      0      0   7.9250        S      1
3       1  female  35.0      1      0  53.1000        S      1
4       3    male  35.0      0      0   8.0500        S      0
```

```
trainDat, testDat = testTrainSplit(datSet, testSize=0.2)
trainDat.shape, testDat.shape
```

```
((713, 8), (178, 8))
```

```
trainDat.head()
```

```
   Pclass     Sex   Age  SibSp  Parch     Fare Embarked  Label
0       3    male  22.0      1      0   7.2500        S      0
2       3  female  26.0      0      0   7.9250        S      1
3       1  female  35.0      1      0  53.1000        S      1
4       3    male  35.0      0      0   8.0500        S      0
5       3    male  28.0      0      0   8.4583        Q      0
```

```
testDat.head()
```

```
     Pclass     Sex   Age  SibSp  Parch     Fare Embarked  Label
101       3    male  28.0      0      0   7.8958        S      0
747       2  female  30.0      0      0  13.0000        S      1
75        3    male  25.0      0      0   7.6500        S      0
```

```
870        3    male  26.0      0      0    7.8958        S      0
700        1  female  18.0      1      0  227.5250        C      1
```

```
datSet.columns
```

```
Index(['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked',
'Label'], dtype='object')
```

```
featType = getFeatType(datSet)
X = datSet.drop(columns=["Label"]).columns
i = 0
for feat in X:
    print(feat, ":", featType[i])
    i += 1
```

```
Pclass : categorical
Sex : categorical
Age : continuous
SibSp : categorical
Parch : categorical
Fare : continuous
Embarked : categorical
```

```
datBelow, datAbove = split(datSet.values, splitFeat=1,
threshold="male")
print(np.unique(datBelow[:, 1]))
print(np.unique(datAbove[:, 1]))
```

```
['female' 'male']
[]
```

```
dat = trainDat.values
getSplits(trainDat.values)
```

```
{0: array([1, 2, 3], dtype=object),
 1: array(['female', 'male'], dtype=object),
 2: [0.71,
  0.79,
  0.875,
  0.96,
  1.5,
  2.5,
  3.5,
  4.5,
  5.5,
  6.5,
  7.5,
  8.5,
  9.5,
  10.5,
  11.5,
```

```
    12.5,
    13.5,
    14.5,
    15.5,
    16.5,
    17.5,
    18.5,
    19.5,
    20.5,
    21.5,
    22.5,
    23.25,
    23.75,
    24.5,
    25.5,
    26.5,
    27.5,
    28.25,
    28.75,
    29.5,
    30.25,
    30.75,
    31.5,
    32.25,
    32.75,
    33.5,
    34.5,
    35.5,
    36.25,
    36.75,
    37.5,
    38.5,
    39.5,
    40.25,
    40.75,
    41.5,
    42.5,
    43.5,
    44.5,
    45.25,
    45.75,
    46.5,
    47.5,
    48.5,
    49.5,
    50.5,
    51.5,
    52.5,
    53.5,
```

```
  54.5,
  55.25,
  55.75,
  56.5,
  57.5,
  58.5,
  59.5,
  60.5,
  61.5,
  62.5,
  63.5,
  64.5,
  67.5,
  70.25,
  70.75,
  72.5,
  77.0],
 3: array([0, 1, 2, 3, 4, 5, 8], dtype=object),
 4: array([0, 1, 2, 3, 4, 5, 6], dtype=object),
 5: [2.00625,
  4.50625,
  5.61875,
  6.36665,
  6.6229,
  6.80415,
  6.91665,
  7.0104,
  7.0479,
  7.052099999999999,
  7.0896,
  7.175,
  7.2271,
  7.239599999999999,
  7.28125,
  7.40415,
  7.5083,
  7.5354,
  7.5896,
  7.6396,
  7.6875,
  7.7271,
  7.731249999999999,
  7.7354,
  7.739599999999999,
  7.74585,
  7.7625,
  7.7854,
  7.7979,
  7.8146,
```

```
    7.8416999999999994,
    7.864599999999999,
    7.8771,
    7.88335,
    7.89165,
    7.9104,
    7.987500000000001,
    8.09375,
    8.1479,
    8.22915,
    8.33125,
    8.38335,
    8.41875,
    8.445799999999998,
    8.5604,
    8.672899999999998,
    8.6979,
    8.78125,
    8.925,
    9.10835,
    9.220849999999999,
    9.2875,
    9.4125,
    9.47915,
    9.49165,
    9.54375,
    9.70625,
    9.83125,
    9.8396,
    9.84375,
    10.0083,
    10.31665,
    10.48125,
    10.81665,
    11.1875,
    11.37085,
    11.75,
    12.1375,
    12.28125,
    12.31875,
    12.4125,
    12.5,
    12.5875,
    12.7625,
    12.9375,
    13.20835,
    13.45835,
    13.64585,
    13.825,
```

```
    13.92915,
    14.2,
    14.4271,
    14.45625,
    14.47915,
    14.75,
    15.0229,
    15.0479,
    15.075,
    15.172899999999998,
    15.3729,
    15.525,
    15.64585,
    15.74585,
    15.8,
    15.875,
    15.95,
    16.05,
    16.4,
    17.049999999999997,
    17.6,
    17.9,
    18.375,
    18.76875,
    19.0229,
    19.37915,
    19.73335,
    20.089599999999997,
    20.23125,
    20.3875,
    20.549999999999997,
    20.7875,
    21.0375,
    21.3771,
    21.8521,
    22.19165,
    22.44165,
    22.7625,
    23.125,
    23.35,
    23.725,
    24.075,
    24.808349999999997,
    25.527099999999997,
    25.75625,
    25.927100000000003,
    25.9646,
    26.125,
    26.26665,
```

```
  26.285400000000003,
  26.3375,
  26.46875,
  26.775,
  27.3604,
  27.7354,
  27.825,
  28.2,
  28.60625,
  28.85625,
  29.0625,
  29.4125,
  29.85,
  30.0354,
  30.2854,
  30.5979,
  30.8479,
  31.1375,
  31.331249999999997,
  31.94375,
  32.75,
  33.25,
  33.760400000000004,
  34.197900000000004,
  34.5146,
  34.8271,
  35.25,
  36.125,
  37.625,
  38.75,
  39.2,
  39.5,
  39.64375,
  39.90625,
  40.8521,
  41.989599999999996,
  44.65,
  47.0,
  48.3,
  49.5021,
  49.7521,
  50.2479,
  50.9875,
  51.67085,
  51.93125,
  52.277100000000004,
  52.8271,
  54.05,
  55.22085,
```

```
    55.67085,
    56.197900000000004,
    56.712500000000006,
    56.964600000000004,
    57.489599999999996,
    58.6896,
    60.287499999999994,
    61.2771,
    61.679199999999994,
    62.66875,
    64.17914999999999,
    65.8,
    67.94999999999999,
    69.425,
    70.275,
    72.25,
    74.375,
    75.77085,
    76.51045,
    77.00835000000001,
    77.62289999999999,
    78.1125,
    78.55834999999999,
    79.025,
    79.42500000000001,
    79.825,
    80.92914999999999,
    82.01455,
    82.66454999999999,
    83.31665,
    84.9875,
    87.8021,
    89.5521,
    90.53960000000001,
    98.7521,
    107.6625,
    109.89165,
    112.07915,
    116.6375,
    126.825,
    134.075,
    135.06664999999998,
    141.07704999999999,
    149.0354,
    152.50625000000002,
    159.1646,
    188.1021,
    211.41875,
    216.6396,
    234.65,
```

```
  254.9479,
  262.6875,
  387.6646],
 6: array(['C', 'Q', 'S'], dtype=object)}
```

```
tree = ID3(trainDat, maxDepth=3)
pprint(tree)
```

```
{'Sex = male': [{'Fare <= 10.81665': [0, {'Age <= 6.5': [1, 0]}]},
                {'Pclass = 3': [{'Fare <= 23.7': [1, 0]}, 1]}]}
```

```
tree = ID3(trainDat, maxDepth=10)
pprint(tree)
```

```
{'Sex = male': [{'Fare <= 10.81665': [{'Fare <= 7.74585': [{'Fare <=
7.239599999999999': [{'Fare <= 7.2271': [{'Fare <= 2.00625': [{'Age <=
26.5': [{'Age <= 22.0': [0,

1]},

0]},

0]},

{'Age <= 21.0': [{'Parch = 1': [0,

1]},

0]}]},

0]},
                                                           {'SibSp =
1': [{'Fare <= 7.8896': [0,

1]},

{'Age <= 28.5': [{'Fare <= 7.7979': [{'Embarked = S': [{'Age <= 19.5':
[0,

{'Age <= 21.5': [1,

0]}]},

0]},

{'Age <= 19.5': [{'Fare <= 8.10415': [{'Fare <= 7.972900000000001':
[0,

{'Age <= 17.0': [0,
```

```
1]}]},

0]},

0]}]},

{'Age <= 32.5': [{'Fare <= 10.0': [{'Fare <= 9.49165': [0,

1]},

0]},

{'Age <= 43.5': [0,

{'Fare <= 9.75': [{'Age <= 45.5': [1,

0]},

1]}]}]}]}]}]}]},
                                      {'Age <= 6.5': [{'Fare <=
20.825': [1,

{'Pclass = 3': [{'Fare <= 31.331249999999997': [0,

{'Fare <= 35.5375': [1,

0]}]},

1]}]},
                                                      {'Pclass = 1':
[{'Age <= 53.0': [{'Fare <= 27.1354': [{'Fare <= 26.14375': [0,

1]},

{'Age <= 27.5': [{'Fare <= 127.81665': [{'Age <= 24.5': [{'Fare <=
95.04165': [0,

1]},

1]},

0]},

{'Fare <= 115.44165000000001': [{'Fare <= 59.0875': [{'Fare <=
56.4146': [0,

1]},

0]},
```

```
{'Age <= 32.0': [0,

1]}]}]}]},

{'Age <= 75.5': [{'Embarked = S': [0,

{'Age <= 57.0': [1,

0]}]},

1]}]},

{'Embarked = C': [{'Parch = 1': [1,

{'Fare <= 12.05835': [1,

{'Fare <= 16.918750000000003': [0,

{'Fare <= 20.23335': [{'Age <= 18.5': [0,

1]},

0]}]}]}]},

{'Fare <= 51.697900000000004': [{'Age <= 9.5': [{'SibSp = 4': [0,

{'Fare <= 41.825': [1,

0]}]},

{'Age <= 27.5': [0,

{'Fare <= 13.25': [0,

{'SibSp = 2': [1,

0]}]}]}]},

{'Fare <= 63.0229': [1,

0]}]}]}]}]}]}]},
                {'Pclass = 3': [{'Fare <= 23.7': [{'Age <= 36.5':
[{'Age <= 1.5': [1,

{'Fare <= 7.8875': [{'Age <= 29.25': [{'Fare <= 7.8021': [{'Fare <=
6.9875': [0,

1]},
```

```
1]},
0]},
{'Fare <= 10.7979': [{'Age <= 19.0': [{'Fare <= 10.1521': [1,
0]},
{'Age <= 25.5': [0,
{'Age <= 27.5': [1,
0]}]}]},
{'Fare <= 13.90835': [1,
{'Fare <= 15.3729': [{'Age <= 28.5': [0,
1]},
1]}]}]}]}]}]},
0]},
                                                  {'Parch = 0': [1,
{'Fare <= 31.331249999999997': [0,
{'Fare <= 32.88125': [1,
0]}]}]}]}]},
                                {'Fare <= 29.35625': [{'Age <= 25.5':
[1,
{'Age <= 27.5': [{'Embarked = S': [0,
1]},
{'Age <= 37.0': [1,
{'Embarked = S': [{'Age <= 39.0': [0,
{'Fare <= 11.75': [{'Age <= 53.5': [1,
0]},
1]}]},
```

```
0]}]}]}]}]},
                                                                        1]}]}]}]}
```

```
eg = testDat.iloc[0]
eg
```

```
Pclass              3
Sex              male
Age              28.0
SibSp               0
Parch               0
Fare           7.8958
Embarked            S
Label               0
Name: 101, dtype: object
```

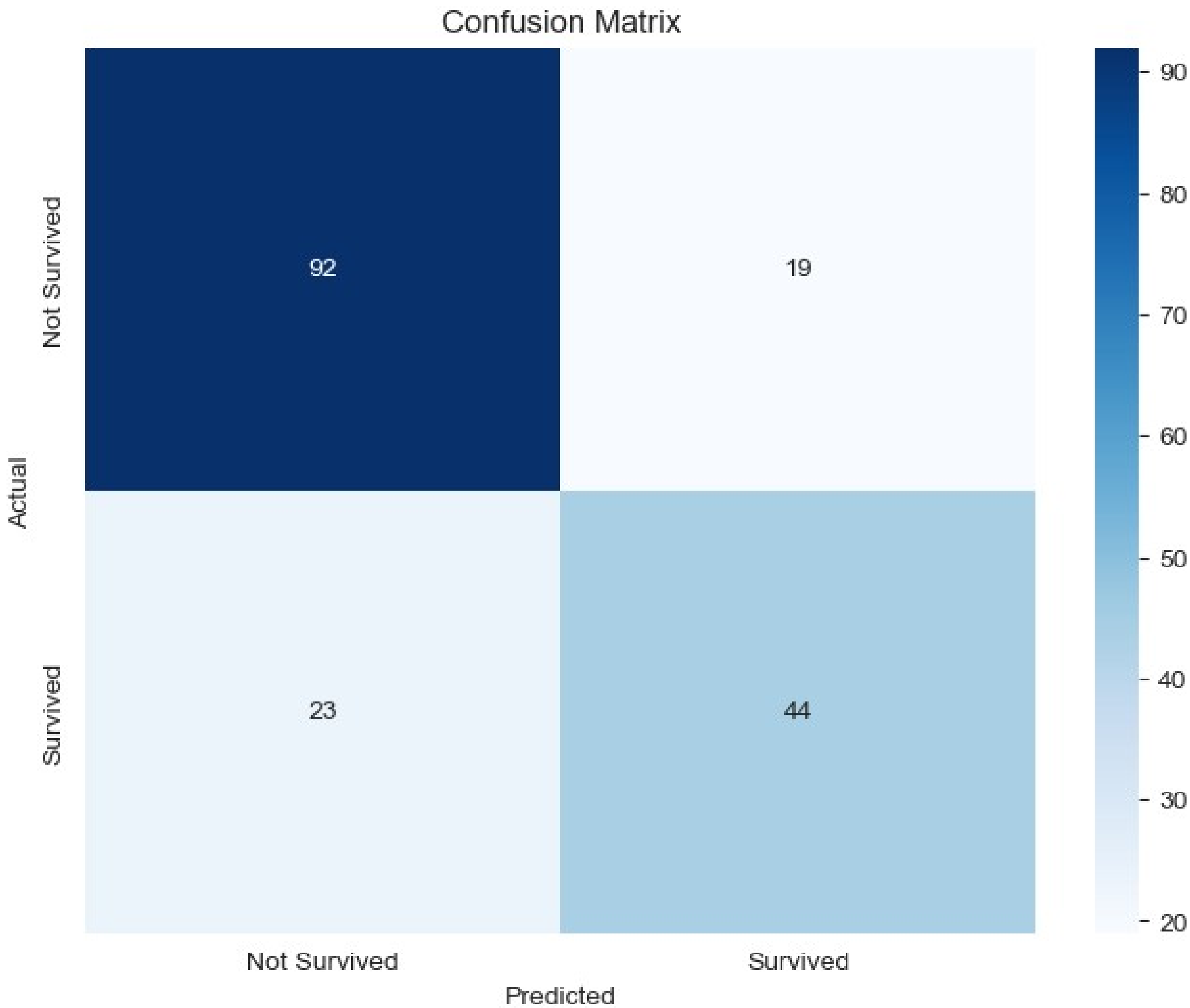
```
classifyEg(eg, tree)
```

```
0
```

```
print(calcAccuracy(testDat, tree))
```

```
0.7640449438202247
```

```
def evaluateModel(tree, test_data, class_names=None):
    test_data = test_data.copy()
    test_data['prediction'] = test_data.apply(lambda row:
classifyEg(row, tree), axis=1)
    y_true = test_data['Label']
    y_pred = test_data['prediction']
    accuracy = (y_true == y_pred).mean()
    report = classification_report(y_true,
y_pred,target_names=class_names,output_dict=True)
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=class_names, yticklabels=class_names)
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title('Confusion Matrix')
    plt.show()
    print(f"Accuracy: {accuracy:.2%}\n")
    print(pd.DataFrame(report).transpose().to_markdown())
    return accuracy, report, cm


accuracy, report, cm = evaluateModel(tree, testDat, class_names=['Not
Survived', 'Survived'])
```



Accuracy: 76.40%

	precision	recall	f1-score	support
Not Survived	0.8	0.828829	0.814159	111
Survived	0.698413	0.656716	0.676923	67
accuracy	0.764045	0.764045	0.764045	0.764045
macro avg	0.749206	0.742773	0.745541	178
weighted avg	0.761762	0.764045	0.762503	178