The goal of this parctical lab is to implement a decision tree classifier

Pseudocode

DecisionTree:

if(stopping condition): retrun decision for this node

1. For each possible feature
2. For each possible split
3. Compute split points
4. Score the split using information gain
5. Take the feature and the split with the best score
6. Split the data points
7. Recurse on each subset

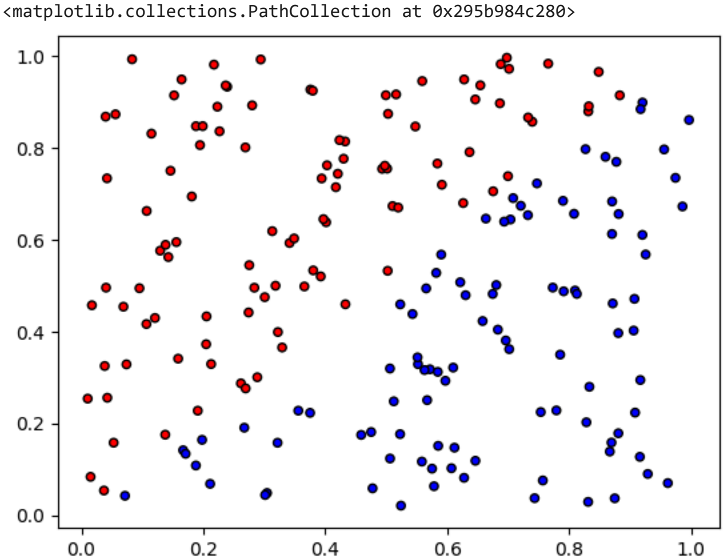
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

```
# Generate 200 2d feature points and their corresponding binary labels.
X = np.random.rand(200, 2)
y = np.zeros(200)
y[np.where(X[:,0]<X[:,1])] = 1
```

```
# Create color maps
cmap_light = ListedColormap(['#AAAAFF', '#FFAAAA'])
cmap_bold = ListedColormap(['#0000FF', '#FF0000'])
```

```
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
            edgecolor='k', s=20)
```

```
<matplotlib.collections.PathCollection at 0x295b984c280>
```




Write a python class called Question used to partition the dataset :

The training data could be seen as a table composed of 3 columns [X, Y] and 200 rows

```
class Question:
    """A Question is used to partition a dataset.
    """

    def __init__(self, column, value):
        self.column = column
        self.value = value
```

```
    def match(self, example):
        # Compare the feature value in an example to the
        # feature value in this question.
        # Since the data is continuous (2D), we use a comparison operator.
        val = example[self.column]
        return val >= self.value
```

For each row in the dataset, check if it matches the question. If so, add it to 'true rows', otherwise, add it to 'false rows'.

```
def split(rows, question):
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows
```

Calculate the gini Impurity or the Entropy

```
def class_counts(rows):
    """Counts the number of each type of example in a dataset.
    Returns:
        dict: A dictionary where keys are unique labels and values are their counts.
    """
    counts = {}
    # The label is the last element in the row.
    # In the setup (X and y), "row" is concatenated like [x1, x2, y]
    # The index for the label is therefore len(row) - 1.
    if len(rows) > 0:
        # Determine the label index assuming uniform structure
        label_index = len(rows[0]) - 1

        for row in rows:
            label = row[label_index]
            if label not in counts:
                counts[label] = 0
            counts[label] += 1
    return counts
```

```
def gini(rows):
    #Class_counts counts the number of each type of example in a dataset.
    counts = class_counts(rows)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(rows))
        impurity -= prob_of_lbl**2
    return impurity
```

Compute the information gain as The uncertainty of the starting node, minus the weighted impurity of two child nodes.

```
def infomation_gain(left, right, current_uncertainty):
    p = float(len(left)) / (len(left) + len(right))
    return current_uncertainty - p * gini(left) - (1 - p) * gini(right)
```

Find the best question to ask by iterating over every feature / value and calculating the information gain

```
def optimal_split(rows):
    best_gain = 0  # keep track of the best information gain
    best_question = None  # keep train of the feature / value that produced it
    current_uncertainty = gini(rows)
    n_features = len(rows[0]) - 1  # number of columns

    for col in range(n_features):  # for each feature

        values = set([row[col] for row in rows])  # unique values in the column

        for val in values:  # for each value
```

```python
            question = Question(col, val)

            # try splitting the dataset
            true_rows, false_rows = split(rows, question)

            # Skip this split if it doesn't divide the
            # dataset.
            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = infomation_gain(true_rows, false_rows, current_uncertainty)

            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question
```

```python
def decisionTree(rows):
    """Recursively builds the decision tree."""

    # Find the best split for the dataset
    gain, question = optimal_split(rows)

    # Stopping condition: If no further information gain, we've reached a leaf.
    if gain == 0:
        # Return the class distribution (Leaf Node)
        return class_counts(rows)

    # Split the data based on the optimal question
    true_rows, false_rows = split(rows, question)

    # Recursively build the true branch.
    true_branch = decisionTree(true_rows)

    # Recursively build the false branch.
    false_branch = decisionTree(false_rows)

    # Return a Decision Node (a tuple containing the question and its branches)
    return (question, true_branch, false_branch)
```

```python
def classify(row, node):
    """Classifies a single row using the built decision tree."""

    if isinstance(node, dict):
        return node

    # a Decision Node is a tuple: (question, true_branch, false_branch)
    question, true_branch, false_branch = node

    if question.match(row):
        return classify(row, true_branch)
    else:
        return classify(row, false_branch)
```

```python

def print_tree(node, spacing=""):

    if isinstance(node, dict):
        print(spacing + "Predict", node)
        return

    question, true_branch, false_branch = node
    print(spacing + str(question))

    print(spacing + '--> True:')
    print_tree(true_branch, spacing + "  ")

    print(spacing + '--> False:')
    print_tree(false_branch, spacing + "  ")
```

## Train and print result

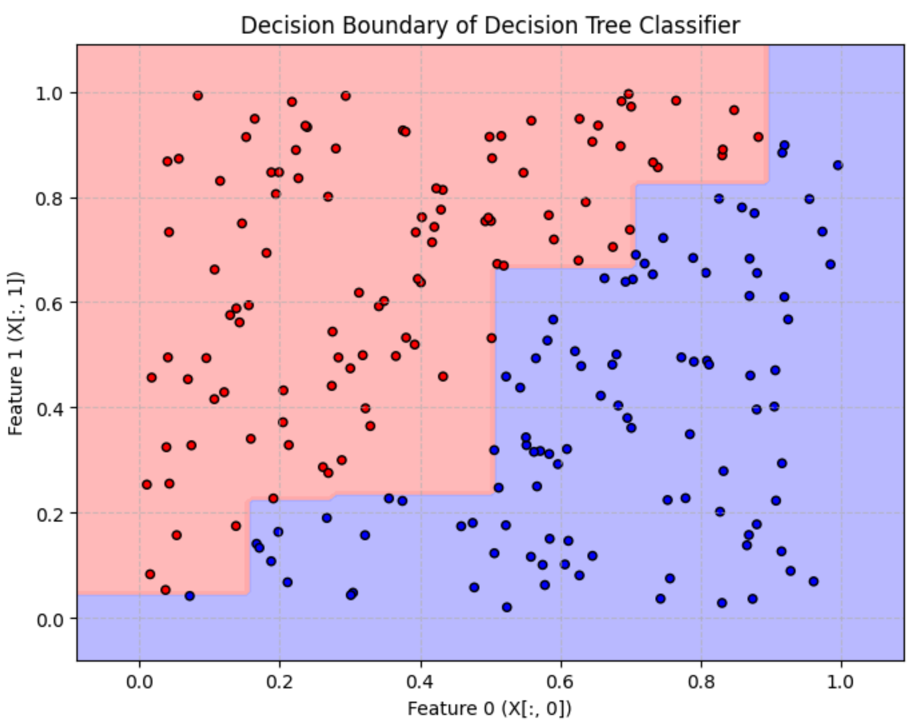
```
# Combine X (features) and y (labels) into a single dataset array
# X has shape (200, 2), y has shape (200,). We reshape y to (200, 1) for hstack.
rows = np.hstack((X, y.reshape(-1, 1)))

my_tree = decisionTree(rows)

print("--- Trained Tree Structure ---")
print_tree(my_tree)
```

```
Building the Decision Tree...

--- Trained Tree Structure ---
Is feature 0 is >=  0.5060?
--> True:
  Is feature 1 is >=  0.6694?
  --> True:
    Is feature 0 is >=  0.7079?
    --> True:
      Is feature 1 is >=  0.8562?
      --> True:
        Is feature 0 is >=  0.9169?
        --> True:
          Predict {np.float64(0.0): 3}
        --> False:
          Predict {np.float64(1.0): 7}
      --> False:
        Predict {np.float64(0.0): 11}
    --> False:
      Predict {np.float64(1.0): 18}
  --> False:
    Predict {np.float64(0.0): 68}
--> False:
  Is feature 1 is >=  0.2536?
  --> True:
    Predict {np.float64(1.0): 73}
  --> False:
    Is feature 0 is >=  0.1669?
    --> True:
      Is feature 1 is >=  0.2273?
      --> True:
        Is feature 1 is >=  0.2274?
        --> True:
          Predict {np.float64(0.0): 1}
        --> False:
          Predict {np.float64(1.0): 1}
      --> False:
        Predict {np.float64(0.0): 13}
    --> False:
      Is feature 1 is >=  0.0536?
      --> True:
        Predict {np.float64(1.0): 4}
      --> False:
        Predict {np.float64(0.0): 1}
```

## using sklearn and matplotlib to visulise the decision boundary

```
from matplotlib.colors import ListedColormap
from sklearn.tree import DecisionTreeClassifier

# --- 1. Prepare and Train the Model (Necessary for the boundary plot) ---

# X and y are assumed to be available from your previous cells.
# We reuse the color maps defined earlier (cmap_light and cmap_bold).
cmap_light = ListedColormap(['#AAAAFF', '#FFAAAA'])
cmap_bold = ListedColormap(['#0000FF', '#FF0000'])

# Train the scikit-learn Decision Tree Classifier
# We use criterion='gini' and no max_depth to match your full training.
clf = DecisionTreeClassifier(criterion='gini', random_state=42)
clf.fit(X, y)

# --- 2. Create the Meshgrid and Prediction Array ---

# Determine plot boundaries (a little padding around the data)
```

```
x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1

# Create a mesh grid of points covering the entire plot area
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

# Predict the class for every point in the mesh grid
# Ravel() flattens the grid, c_[] stacks the coordinates, and reshape() returns it to grid shape.
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# --- 3. Plot the Decision Boundary and Data Points ---

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

# Plot the decision boundary (colored regions)
# The boundary is where the color changes from blue to red in the contour plot.
plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.8)

# Plot the original training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
            edgecolor='k', s=20)

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("Decision Boundary of Decision Tree Classifier")
plt.xlabel("Feature 0 (X[:, 0])")
plt.ylabel("Feature 1 (X[:, 1])")
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```




using sklearn to generate a better graph of the the decision Tree, i put the image with the code

```
from sklearn.tree import DecisionTreeClassifier, export_graphviz

# 1. Prepare data for scikit-learn
# X and y are assumed to be available from your previous cells.
# X is the features (200, 2), y is the labels (200,)

# 2. Train a scikit-learn Decision Tree Classifier
# We set max_depth to a small value (like 3) for a cleaner visualization,
# but using the default depth might better reflect your custom implementation
```

```
# but using the default depth might better reflect your custom implementation.
# criterion='gini' matches the impurity metric you used.
print("Training scikit-learn Decision Tree (Criterion: Gini)...")
clf = DecisionTreeClassifier(criterion='gini', random_state=42)
clf.fit(X, y)

# 3. Export the tree structure to the DOT format
print("\n--- Decision Tree Structure (DOT format for graphviz) ---")
print("Copy this output and paste it into an online graphviz renderer (e.g., https://dreampuf.github.io/GraphvizOnline/) to see

# We use export_graphviz to generate the DOT string.
dot_data = export_graphviz(
    clf,
    out_file=None,
    feature_names=['Feature 0', 'Feature 1'], # Your two columns
    class_names=['Class 0.0', 'Class 1.0'],   # Your two labels
    filled=True,
    rounded=True,
    special_characters=True
)

print(dot_data)
```

```
Training scikit-learn Decision Tree (Criterion: Gini)...

--- Decision Tree Structure (DOT format for graphviz) ---
Copy this output and paste it into an online graphviz renderer (e.g., https://dreampuf.github.io/GraphvizOnline/) to see the vis
digraph Tree {
node [shape=box, style="filled, rounded", color="black", fontname="helvetica"] ;
edge [fontname="helvetica"] ;
0 [label=<Feature 0 &le; 0.504<br/>gini = 0.5<br/>samples = 200<br/>value = [97, 103]<br/>class = Class 1.0>, fillcolor="#f3f9fd
1 [label=<Feature 1 &le; 0.241<br/>gini = 0.271<br/>samples = 93<br/>value = [15, 78]<br/>class = Class 1.0>, fillcolor="#5fb0ea
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
2 [label=<Feature 0 &le; 0.152<br/>gini = 0.375<br/>samples = 20<br/>value = [15, 5]<br/>class = Class 0.0>, fillcolor="#eeab7b"
1 -> 2 ;
3 [label=<Feature 1 &le; 0.048<br/>gini = 0.32<br/>samples = 5<br/>value = [1, 4]<br/>class = Class 1.0>, fillcolor="#6ab6ec"] ;
2 -> 3 ;
4 [label=<gini = 0.0<br/>samples = 1<br/>value = [1, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
3 -> 4 ;
5 [label=<gini = 0.0<br/>samples = 4<br/>value = [0, 4]<br/>class = Class 1.0>, fillcolor="#399de5"] ;
3 -> 5 ;
6 [label=<Feature 1 &le; 0.225<br/>gini = 0.124<br/>samples = 15<br/>value = [14, 1]<br/>class = Class 0.0>, fillcolor="#e78a47"
2 -> 6 ;
7 [label=<gini = 0.0<br/>samples = 13<br/>value = [13, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
6 -> 7 ;
8 [label=<Feature 0 &le; 0.273<br/>gini = 0.5<br/>samples = 2<br/>value = [1, 1]<br/>class = Class 0.0>, fillcolor="#ffffff"] ;
6 -> 8 ;
9 [label=<gini = 0.0<br/>samples = 1<br/>value = [0, 1]<br/>class = Class 1.0>, fillcolor="#399de5"] ;
8 -> 9 ;
10 [label=<gini = 0.0<br/>samples = 1<br/>value = [1, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
8 -> 10 ;
11 [label=<gini = 0.0<br/>samples = 73<br/>value = [0, 73]<br/>class = Class 1.0>, fillcolor="#399de5"] ;
1 -> 11 ;
12 [label=<Feature 1 &le; 0.663<br/>gini = 0.358<br/>samples = 107<br/>value = [82, 25]<br/>class = Class 0.0>, fillcolor="#eda7
0 -> 12 [labeldistance=2.5, labelangle=-45, headlabel="False"] ;
13 [label=<gini = 0.0<br/>samples = 68<br/>value = [68, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
12 -> 13 ;
14 [label=<Feature 0 &le; 0.705<br/>gini = 0.46<br/>samples = 39<br/>value = [14.0, 25.0]<br/>class = Class 1.0>, fillcolor="#a8
12 -> 14 ;
15 [label=<gini = 0.0<br/>samples = 18<br/>value = [0, 18]<br/>class = Class 1.0>, fillcolor="#399de5"] ;
14 -> 15 ;
16 [label=<Feature 1 &le; 0.826<br/>gini = 0.444<br/>samples = 21<br/>value = [14, 7]<br/>class = Class 0.0>, fillcolor="#f2c09c
14 -> 16 ;
17 [label=<gini = 0.0<br/>samples = 11<br/>value = [11, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
16 -> 17 ;
18 [label=<Feature 0 &le; 0.9<br/>gini = 0.42<br/>samples = 10<br/>value = [3, 7]<br/>class = Class 1.0>, fillcolor="#8ec7f0"] ;
16 -> 18 ;
19 [label=<gini = 0.0<br/>samples = 7<br/>value = [0, 7]<br/>class = Class 1.0>, fillcolor="#399de5"] ;
18 -> 19 ;
20 [label=<gini = 0.0<br/>samples = 3<br/>value = [3, 0]<br/>class = Class 0.0>, fillcolor="#e58139"] ;
18 -> 20 ;
}
```