Decision trees (60 points total)

Below is the code for a decision tree classifier.

In [25]:
```
import math
import scipy.stats
import random
from scipy.stats import chi2_contingency
from random import sample


ENABLE_PRUNING = False
RANDOM_FOREST = False
PRUNE_THRESHOLD = 0.05  # Threshold for p-value in chi-square test for pruni
RANDOM_DECISIONS_RATIO = 0.3  # Fraction of decisions to consider if RANDOM_

class DecisionTree:
  """ A decision tree for machine learning.  Since it's a tree,
  it's defined as a node with possible subtrees as children.

  self.leaf (boolean):  Whether the node is a leaf (no children).
  self.outcome (boolean):  If this is a leaf, its recommended classificatior
     a classified example that ends up there.
  self.decision (Decision):  If this isn't a leaf, the decision represented
     the node.  See Decision class below.
  self.yes (DecisionTree):  The subtree followed if an example answers "yes"
     to the decision.
  self.no (DecisionTree):  The subtree followed if an example answers "no"
     to the decision.
  """

  def __init__(self, outcome):
    """A constructor for a leaf."""
    self.leaf = True
    self.outcome = outcome
    self.decision = None
    self.yes = None
    self.no = None

  def __init__(self, decision, yes, no):
    """A constructor for an interior node."""
    self.leaf = False
    self.decision = decision
    self.yes = yes
    self.no = no

  # Examples are assumed to be a list-of-lists with each list
  # an example.
  def __init__(self, examples, labels):
    """A recursive constructor for building the tree from examples."""
    agree, label = all_agree(labels)
    if (agree):
```

```
      self.leaf = True
      self.outcome = label
      self.decision = None
      self.yes = None
      self.no = None
      return
    all_decisions = generate_decisions(examples)
    if RANDOM_FOREST:
        decisions_sample_size = int(math.sqrt(len(all_decisions)))
        all_decisions = random.sample(list(all_decisions), decisions_sample_
    best_decision = None
    best_entropy = 1
    best_split = None
    for decision in all_decisions:
      split = split_by_decision(decision, examples, labels)
      expected_entropy = try_split(split)
      if expected_entropy < best_entropy:
        best_decision = decision
        best_entropy = expected_entropy
        best_split = split
    # Check whether nothing improved - we didn't split
    if best_split == None or len(best_split.yes_examples) == 0 or len(best_s
      self.leaf = True
      self.outcome = majority(labels)
      self.decision = None
      self.yes = None
      self.no = None
      return

    self.leaf = False
    self.outcome = None
    self.decision = best_decision
    self.yes = DecisionTree(best_split.yes_examples, best_split.yes_labels)
    self.no = DecisionTree(best_split.no_examples, best_split.no_labels)
    if ENABLE_PRUNING and self.prune(best_split):
      self.leaf = True
      self.outcome = majority(labels)
      self.decision = None
      self.yes = None
      self.no = None
    return

  def __str__(self):
    return recursive_string(self, 0)

  # TODO
  def prune(self, best_split):
    """No effect (return False) unless both children are leaves.
    If they are, return True if a chi-square test between feature
    and label is not significant -- the caller will prune the node,
    turning it into a leaf."""
    """Implements pruning using a chi-square test."""
    if not self.yes.leaf or not self.no.leaf:
        # If either child is not a leaf, do not prune
        return False
```

```
    # Calculate counts for the chi-square test
    yes_label_count = sum(best_split.yes_labels)
    no_label_count = sum(best_split.no_labels)
    yes_feature_count = len(best_split.yes_labels)
    no_feature_count = len(best_split.no_labels)

    # Counts for [[featureANDlabel, featureANDNOTlabel],[NOTfeatureANDlabel,
    feature_and_label = yes_label_count
    feature_and_not_label = yes_feature_count - yes_label_count
    not_feature_and_label = no_label_count
    not_feature_and_not_label = no_feature_count - no_label_count

    contingency_table = [[feature_and_label, feature_and_not_label], [not_fe

    # Perform chi-square test
    _, p_value, _, _ = chi2_contingency(contingency_table)

    # If p-value < 0.05, the feature and label are considered dependent, and
    # If p-value >= 0.05, they are independent, and we can prune
    return p_value >= 0.05

  def classify(self, example):
    """Recursively decides how this tree would classify the passed example."
    if self.leaf:
      return self.outcome
    if self.decision.applies_to(example):
      return self.yes.classify(example)
    return self.no.classify(example)

def recursive_string(tree, indent):
    """Recursively print the tree with an indentation corresponding to
    tree depth.  Useful for debugging.  Is also the __str__() implementation
    if (tree.leaf):
      return '  ' * indent + str(tree.outcome) + '\n'
    else:
      mystr = '  ' * indent + 'if ' + str(tree.decision) + ':\n'
      mystr += recursive_string(tree.yes, indent+1)
      mystr += recursive_string(tree.no, indent+1)
      return mystr

# Assume numerical features for convenience
class Decision:
  """Object representing a decision to make about an example.  Each interior
  node of the tree has one of these.
  feature_num:  Index into which feature is being used for the decision.
  thresh:  For features that are numeric, the numerical threshold for return
  """

  def __init__(self, feature_num, thresh):
    self.feature_num = feature_num
    self.thresh = thresh

  def applies_to(self, example):
    """Returns true if the example should follow the "yes" branch for the de
    if example[self.feature_num] >= self.thresh:
      return True
```

```
    return False

  def __str__(self):
    return "Feature " + str(self.feature_num) + " >= " + str(self.thresh)

# Split carries yes examples, yes labels, no examples, no labels
# for convenience
class Split:
  """If a Decision would separate the examples into two piles, then a Split
  represents those two piles.

  yes_examples(list-of-lists):  The examples that would satisfy the Decision
  yes_labels(list of bools):  The labels on the yes_examples.
  no_examples(list-of-lists):  The examples that don't satisfy the Decision.
  no_labels(list of bools):  The labels of the no_examples."""
  def __init__(self, yes_examples, yes_labels, no_examples, no_labels):
    self.yes_examples = yes_examples
    self.yes_labels = yes_labels
    self.no_examples = no_examples
    self.no_labels = no_labels

  # For debugging
  def __str__(self):
    out = str(self.yes_examples) + '\n'
    out += str(self.yes_labels) + '\n'
    out += str(self.no_examples) + '\n'
    out += str(self.no_labels) + '\n'
    return out

def majority(labels):
  """Determine whether the majority of the labels is 1 (return True) or 0 (F
  yes_count = sum(labels)
  if yes_count >= len(labels)/2:
    return True
  return False

def all_agree(labels):
  """First return value is whether all the labels are the same.
  Second return value is the majority classification of the labels."""
  return (sum(labels) == len(labels)) or (sum(labels) == 0), majority(labels

def generate_decisions(examples):
  """Given a list of examples, generate all possible Decisions based on thos
  examples' features and numerical values.  Return a list of those Decisions
  decisions = set() # Use set to avoid decision duplication
  feature_count = len(examples[0])
  for example in examples:
    for j in range(feature_count):
      decisions.add(Decision(j,example[j]))
  return decisions

def try_split(split):
  """Given the split of examples that did and didn't satisfy the Decision,
  calculate the expected entropy (the criterion used to find the best Decisi
  yes_entropy = entropy(split.yes_labels)
  no_entropy = entropy(split.no_labels)
```

```
  example_count = len(split.yes_labels) + len(split.no_labels)
  yes_prob = len(split.yes_labels)/example_count
  no_prob = len(split.no_labels)/example_count
  expected = yes_prob * yes_entropy + no_prob * no_entropy
  return expected

def split_by_decision(decision, examples, labels):
  """Using the Decision argument, divide the examples into those that satisf
  the Decision and those that don't, and create a Split object to keep these
  two piles separate.  Split the corresponding labels as well."""
  yes_examples = []
  yes_labels = []
  no_examples = []
  no_labels = []
  for i, example in enumerate(examples):
    if example[decision.feature_num] >= decision.thresh:
      yes_examples += [example]
      yes_labels += [labels[i]]
    else:
      no_examples += [example]
      no_labels += [labels[i]]
  return Split(yes_examples, yes_labels, no_examples, no_labels)

def entropy(bool_list):
  """Given a list of True and False values (or 0's and 1's), calculate the
  entropy of the list."""
  true_count = sum(bool_list)
  false_count = len(bool_list) - sum(bool_list)
  if true_count == 0 or false_count == 0:
    return 0
  true_prob = true_count/len(bool_list)
  false_prob = false_count/len(bool_list)
  return - true_prob * math.log(true_prob, 2) - false_prob * math.log(false_
```

Upload 'adult2000.csv' with the following code. This is census data where the target variable to predict is whether the person made $50K/year or more. (We're just using the first 2000 entries for speed reasons.)

In [17]:
```
import pandas as pd
```

In [3]:
```
df = pd.read_csv('adult2000.csv')
df.head()
```

Out[3]:

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male
2	38	Private	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male
3	53	Private	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male
4	28	Private	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female

The "Target" column has our labels, whether the individual made $50K/year or more.

In [4]:

```
labels = df["Target"]
labels
```

Out[4]:

```
0       0
1       0
2       0
3       0
4       0
       ..
1994    1
1995    0
1996    1
1997    0
1998    1
Name: Target, Length: 1999, dtype: int64
```

Since the decision tree code only works with numerical data, we can turn the string data into numerical data using "one-hot encoding". We make a new column for each possible value of the categorical data, and use True and False values for that column, which are interpreted later in the code as 1's and 0's. (Note: if you are frustrated by how long it takes to train your decision tree, you could temporarily skip this cell when loading the data and just use num_features in the cell that follows. Be sure to revert this before turning the assignment in.)

In [5]:

```
def one_hot(df, colname):
    values = df[colname].unique()
    for value in values:
        df[value] = df[colname] == value
    return df

one_hot(df, "workclass")
```

```
one_hot(df, "marital-status")
one_hot(df, "occupation")
one_hot(df, "relationship")
one_hot(df, "race")
one_hot(df, "sex")
one_hot(df, "native-country")
```

Out[5]:

	age	workclass	education	education-num	marital-status	occupation	relationship	race	s
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	M
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	M
2	38	Private	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	M
3	53	Private	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	M
4	28	Private	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Fem
...	...	...	...	...	...	...	...	...	
1994	30	Private	Bachelors	13	Never-married	Exec-managerial	Unmarried	White	Fem
1995	44	Private	Some-college	10	Divorced	Machine-op-inspct	Not-in-family	White	M
1996	49	Private	HS-grad	9	Married-civ-spouse	Sales	Husband	White	M
1997	75	Self-emp-not-inc	Masters	14	Married-spouse-absent	Prof-specialty	Not-in-family	White	Fem
1998	37	Private	Bachelors	13	Married-civ-spouse	Sales	Husband	White	M

1999 rows × 92 columns

In [6]:

```
num_features = df[["age", "education-num", "capital-gain", "capital-loss", "
one_hot_features = df.iloc[:, 14:]
features = pd.concat([num_features, one_hot_features], axis=1)
features.head()
```

Out[6]:

	age	education-num	capital-gain	capital-loss	hours-per-week	State-gov	Self-emp-not-inc	Private	Federal-gov	Local-gov	...
0	39	13	2174	0	40	True	False	False	False	False	...
1	50	13	0	0	13	False	True	False	False	False	...
2	38	9	0	0	40	False	False	True	False	False	...
3	53	7	0	0	40	False	False	True	False	False	...
4	28	13	0	0	40	False	False	True	False	False	...

5 rows × 83 columns

In [7]:

```
from sklearn.model_selection import train_test_split

features_train, features_test, labels_train, labels_test = train_test_split(

features_train_list = features_train.values.tolist()
labels_train_list = labels_train.tolist()
features_test_list = features_test.values.tolist()
labels_test_list = labels_test.tolist()
```

With list-format train/test split in hand, now it's time to train and evaluate a model.

(1) (2 points) In the code box below, train a model on the adult2000 training data using the DecisionTree constructor we built above.

In [8]:

```
# TODO make a decision tree!
# Proceeding without converting labels_train and labels_test as they are alr
features_train_list = features_train.values.tolist()
features_test_list = features_test.values.tolist()

# Initialize and train the DecisionTree model
model = DecisionTree(features_train_list, labels_train_list)
print("Decision Tree model training complete.")

# Assuming the DecisionTree class has a method for classification (predictio
# Let's pretend we have a method called classify for individual predictions

def evaluate_model(model, features_test, labels_test):
    correct_predictions = 0
    for i, test_example in enumerate(features_test):
        prediction = model.classify(test_example)
        if prediction == labels_test[i]:
            correct_predictions += 1
    accuracy = correct_predictions / len(labels_test)
    return accuracy

# Evaluate the model's accuracy on the test data
accuracy = evaluate_model(model, features_test_list, labels_test_list)
print(f"Model accuracy on test data: {accuracy * 100:.2f}%")
```

```
Decision Tree model training complete.
Model accuracy on test data: 82.80%
```

(2) (6 points) Code a function evaluate() that takes a trained DecisionTree, a set of examples, and a set of labels, and returns an accuracy, the number of examples it got right. Note that it should return perfect accuracy on the test below, and should return above 90% accuracy on the training set for the adult2000 dataset.

In [9]:
```python
# Returns accuracy - TODO
def evaluate(tree, test_examples, test_labels):
    correct_predictions = 0
    total_examples = len(test_examples)

    for i, example in enumerate(test_examples):
        # Predict the label for each test example
        predicted_label = tree.classify(example)
        # Compare the predicted label with the actual label
        if predicted_label == test_labels[i]:
            correct_predictions += 1

    # Calculate accuracy as the ratio of correct predictions to total exampl
    accuracy = correct_predictions / total_examples
    return accuracy

# Test - expect perfect accuracy
test_examples = [[0,0],[0,1],[1,0],[1,1]]
test_labels = [1, 1, 0, 0]
test_tree = DecisionTree(test_examples,test_labels)
evaluate(test_tree, test_examples, test_labels)
```

Out[9]:
```
1.0
```

(3) (4 points) Use your evaluate function to print your trained model's accuracy for both the training data and the test data. Then explain in your own words below why one accuracy is much higher than the other.

In [11]:
```python
# Evaluate on training data
train_accuracy = evaluate(model, features_train_list, labels_train_list)
print(f"Training Data Accuracy: {train_accuracy * 100:.2f}%")

# Evaluate on test data
test_accuracy = evaluate(model, features_test_list, labels_test_list)
print(f"Test Data Accuracy: {test_accuracy * 100:.2f}%")
```

```
Training Data Accuracy: 99.67%
Test Data Accuracy: 82.80%
```

TODO why is training score so much higher than test?

Explaination:

The discrepancy between the high training accuracy (99.67%) and lower test accuracy (82.80%) with the Decision Tree model suggests overfitting.

To address overfitting and improve the model's generalization to new data, we can:

1. Pruning the decision tree to remove less critical parts.
2. Implementing cross-validation to better estimate model performance on unseen data.
3. Using ensemble methods like Random Forests, which can reduce overfitting by averaging multiple decision trees.
4. Adopting these strategies can help balance the model's performance, enhancing its accuracy on both training and unseen datasets.

(4, 6 pts) The following tiny dataset seems very similar to the test dataset that we got perfect accuracy on, a couple code boxes back. But, it doesn't seem to produce perfect accuracy. Is it possible to design by hand a decision tree that gets perfect accuracy on this dataset? If so, why doesn't our code succeed in automatically constructing it? If not, why is perfect classification not possible here?

In [12]:
```
# What's happening here?
test_examples = [[0,0],[0,1],[1,0],[1,1]]
test_labels = [0, 1, 1, 0]
test_tree = DecisionTree(test_examples,test_labels)
evaluate(test_tree, test_examples, test_labels)
```

Out[12]: 0.5

TODO

The dataset represents an XOR logic function, which cannot be perfectly classified by a simple, linear decision tree because XOR requires a non-linear decision boundary. While it's theoretically possible to design a decision tree by hand that perfectly classifies the XOR dataset by creating a multi-level structure, automatic construction might fail due to:

Linear Separability: XOR is not linearly separable with a single decision boundary, challenging for simple decision trees that rely on linear decisions at each node. Model Complexity: Perfectly classifying XOR requires a tree with at least two levels to capture the feature interaction, which might not be achieved if the model complexity is restricted. In essence, the XOR problem's complexity exceeds the capacity of basic decision trees without manual feature engineering or allowing for a sufficiently complex tree structure.

(5) (8 points) One way to avoid overfitting in decision trees is pruning, or getting rid of decisions that aren't pulling their weight. Complete the function prune() that returns true if the node should be pruned to be a leaf. The function should first check whether both children are leaves; if not, the node is safe from pruning. If both nodes are leaves, the code should check whether the best decision's feature and the label are independent according to a chi-square test, and if so, return True. You can use the function scipy.stats.chi2_contingency(), where the four cells in the list-of-lists provided as input are counts of [[featureANDlabel, featureANDNOTlabel], [NOTfeatureANDlabel,NOTfeatureANDNOTlabel]]. (Hint: You can compute these counts from just the yes_label and no_label lists in a Split object.) The p-value of the contingency test (second return value) must be < 0.05 to keep the decision.

When it works, you should see the train and test accuracies be more similar to each other. The test accuracy should be a little higher or similar (randomness plays a part in the results).

In [18]:

```
ENABLE_PRUNING = True
my_tree = DecisionTree(features_train_list,labels_train_list)
print(evaluate(my_tree, features_train_list, labels_train_list))
print(evaluate(my_tree, features_test_list, labels_test_list))
```

```
0.8965977318212142
0.828
```

Next we'll see whether turning this into an ensemble learner helps at all. We'll try turning the single tree into a random forest.

(6, 8 pts) Code the function bag(), which takes a list of N examples and N labels and returns a list of N examples and N corresponding labels that have been sampled with replacement from the original lists. (You'll find random.randrange() helpful for this part.)

In [28]:

```
from random import randrange

def bag(examples, labels):
    new_examples = []
    new_labels = []
    N = len(examples)

    for _ in range(N):
        index = randrange(N)  # Randomly select an index
        new_examples.append(examples[index])  # Add the example at that inde
        new_labels.append(labels[index])  # Add the corresponding label to t

    return new_examples, new_labels
```

(7, 7 pts) Code a RandomForest constructor that takes the lists of N examples and N labels, as well as an argument for the number of random trees, and creates the list of decision trees that could be used to vote on the correct classification. You don't need to sample features at each node, but can use the decision tree constructor you already have.

(8, 3 pts) Add a few lines to the original decision tree method where, if the RANDOM_FOREST variable is true, all_decisions uses a random sample of its decisions of length sqrt(len(all_decisions)). (You'll find the function random.sample() handy here.)

(9, 6 pts) Code a classify() method for your RandomForest that asks its decision trees to vote on a classification, and returns their majority decision.

In [29]:
```
class RandomForest:

  def __init__(self, examples, labels, tree_count):
        self.trees = []
        for _ in range(tree_count):
            # Create a bootstrapped sample for each tree
            sample_examples, sample_labels = bag(examples, labels)
            # Create a tree from each sample
            tree = DecisionTree(sample_examples, sample_labels)
            self.trees.append(tree)

  def classify(self, example):
        # Collect votes from all trees
        votes = [tree.classify(example) for tree in self.trees]
        # Majority voting
        majority_vote = max(set(votes), key=votes.count)
        return majority_vote
```

(10, 4 pts) Get the new train and test accuracies for your RandomForest, using either your original evaluate function or a similar one. The number of trees to create is up to you, but you should at least get similar performance to the single tree with pruning. You can turn off pruning to speed things up slightly.

In [30]:
```
ENABLE_PRUNING = False
RANDOM_FOREST = True
# TODO
num_trees = 10
my_forest = RandomForest(features_train_list, labels_train_list, num_trees)

# Now, let's use the evaluate function to get accuracies
train_accuracy = evaluate(my_forest, features_train_list, labels_train_list)
test_accuracy = evaluate(my_forest, features_test_list, labels_test_list)

print(f"RandomForest Training Data Accuracy: {train_accuracy * 100:.2f}%")
print(f"RandomForest Test Data Accuracy: {test_accuracy * 100:.2f}%")
```

```
RandomForest Training Data Accuracy: 93.06%
RandomForest Test Data Accuracy: 83.20%
```

(11, 6 pts) One last "thought question." Suppose we want to train a decision tree to just say "yes" to one datapoint and "no" to all other points in the data. Assume all features are continuous. How many decision nodes (not leaf nodes) are necessary, as a function of the number of continuous features n, to make this decision tree say "yes" to a high-dimensional cube around the target point, and "no" to all points outside the hypercube? And what is the rough shape of the tree that does this?

TODO

To create a decision tree that says "yes" only to one specific data point within a high-dimensional space of n continuous features, and "no" to all others, we need $2n$ decision nodes. This is because for each continuous feature, we require two decision nodes: one to set the lower boundary and another for the upper boundary of the feature value, effectively isolating the target point within a hypercube.

Rough Shape of the Tree: The tree will have a highly unbalanced or linear shape, with a sequence of decision nodes for each feature that sequentially checks the lower and upper bounds. This forms a path leading to a "yes" for the target point, while all other paths lead to "no".

In summary: Number of Decision Nodes Required: $2n$ (for n features). Tree Shape: A linear sequence of decisions, highly unbalanced, leading to a hypercube around the target point. This approach highlights an extreme case of overfitting, creating a model that is overly complex and specific to the training data, with poor generalizability to new data.

When you're done, use "File->Download .ipynb" and upload your .ipynb file to Blackboard, along with a PDF version (File->Print->Save as PDF) of your assignment.