Decision trees (60 points total)

Below is the code for a decision tree classifier.

```
import math
import scipy.stats
import random

ENABLE_PRUNING = False
RANDOM_FOREST = False

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 classification of
     a classified example that ends up there.
  self.decision (Decision):  If this isn't a leaf, the decision represented by
     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:
      # TODO sample all_decisions
      all_decisions = random.sample(all_decisions, math.floor(math.sqrt(len(all_decisions))))
    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_split.no_examples) == 0:
      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."""
    # TODO
    if (self.yes != None and self.no != None and self.yes.leaf and self.no.leaf):
      yesAndLabel = sum(1 if x else 0 for x in best_split.yes_labels)
      yesNoLabel = len(best_split.yes_labels) - yesAndLabel
      noAndLabel = sum(1 if x else 0 for x in best_split.no_labels)
      noNoLabel = len(best_split.no_labels) - noAndLabel
      res = scipy.stats.chi2_contingency([[yesAndLabel, yesNoLabel],[noAndLabel, noNoLabel]])
      if res.pvalue >= 0.05:
        return True
    return False

  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 returning True.
  """

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

  def applies_to(self, example):
    # print("applies to: " + str(example) + ", feature_num: " + str(self.feature_num))
    """Returns true if the example should follow the "yes" branch for the decision."""
    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 (False)."""
    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 those
    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 Decision)."""
    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 satisfy
    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_prob,2)
```

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.)

```
import pandas as pd


# Google Colab only cell
from google.colab import files
import io

uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving adult2000.csv to adult2000.csv
```

```
df = pd.read_csv('adult2000.csv')
df.head()
```

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	Target
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	0
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	0
2	38	Private	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	0
3	53	Private	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	0
4	28	Private	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	0

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

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

```
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.)

```
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")
```

	age	workclass	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	...	Guatemala
0	39	State-gov	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	...	False
1	50	Self-emp-not-inc	Bachelors	13	Married-civ-	Exec-managerial	Husband	White	Male	0	...	False

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

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

5 rows × 83 columns

Married-

```
from sklearn.model_selection import train_test_split

features_train, features_test, labels_train, labels_test = train_test_split(features,labels, random_state=110)

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()


# print(features_train_list)
# print(features_test_list)
```

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.

```
# TODO make a decision tree! (usually takes 1 min to train)
ENABLE_PRUNING = False
decision_tree = DecisionTree(features_train_list, labels_train_list)
```

(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.

```
from numpy import double
# Returns accuracy - TODO
def evaluate(tree, test_examples, test_labels):
  success_count = 0
  for i in range(len(test_examples)):
    if tree.classify(test_examples[i]) == test_labels[i]:
      success_count += 1
  # print(success_count)
  # print(len(test_examples))
  return float(success_count)/len(test_examples)

# 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)
print(evaluate(test_tree, test_examples, test_labels))
print(evaluate(decision_tree, features_train_list, labels_train_list))
```

```
1.0
0.9966644429619747
```

(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.

```
# TODO evaluate on train and test data
print(evaluate(decision_tree, features_train_list, labels_train_list))
print(evaluate(decision_tree, features_test_list, labels_test_list))
```

```
0.9966644429619747
0.828
```

TODO why is training score so much higher than test?

The training score is so much higher than the test because the decision tree was trained to identify questions that minimize the entropy on the training data. Due to the fact that the decision tree was trained to minimize the entropy of the training data as much as possible, it reaches a near perfect performance. The testing dataset has data that the decision tree hasn't seen before, and therefore it won't do quite as well in terms of reaching correct decisions.

(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?

```
# 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)
```

```
0.5
```

TODO It is possible to design by hand a decision tree that gets perfect accuracy on this dataset. Looking at the examples and labels, it can be identified that the correct label for an example is the XOR of the two values. Our code doesn't succeed in automatically constructing it because our current implementation looks at one feature at a time, rather than comparing the values of multiple features together (as required by an XOR operation).

(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).

```
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.)

```
from random import randrange

def bag(examples, labels):
  # TODO
  new_examples = []
  new_labels = []
  while(len(new_examples) < len(examples)):
    rand_index = random.randrange(0, len(examples))
    new_examples.append(examples[rand_index])
    new_labels.append(labels[rand_index])
  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.

```
class RandomForest:

  def __init__(self, examples, labels, tree_count):
    # TODO
    self.trees = []
    for i in range(tree_count):
      bag_examples, bag_labels = bag(examples, labels)
      self.trees.append(DecisionTree(bag_examples, bag_labels))
    return

  def classify(self, example):
    # TODO
    counts = {0: 0, 1: 0}
    for tree in self.trees:
      counts[tree.classify(example)] += 1
    return max(counts, key=counts.get)
```

(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.

```
ENABLE_PRUNING = False
RANDOM_FOREST = True
# TODO
r_forest = RandomForest(features_train_list, labels_train_list, 10)
```

```
<ipython-input-32-85647ae39d0f>:53: DeprecationWarning: Sampling from a set deprecated
since Python 3.9 and will be removed in a subsequent version.
  all_decisions = random.sample(all_decisions, math.floor(math.sqrt(len(all_decisions))))
```

```
# evaluate
print(evaluate(r_forest, features_test_list, labels_test_list))
```

```
0.846
```

(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 I believe that the number of decision nodes necessary to make the decision tree say "yes" to a high-dimensional cube around the target point and "no" to all points outside the hypercube would be $2n$. Since we want the decision tree to say "yes" in a very small range around the target point, we would want one decision to check if a certain feature is less than a certain value and another decision to check if the feature is

greater than a certain value. These two decisions would be needed for every feature in order to ensure that all features are within the high-dimensional cube of "yes."

The shape of the tree would look like a long line of $2n$ decision nodes. There would be a leaf node coming off of each of the decision nodes (2 for the last decision node) which would return "no" if the point is outside of the hypercube for a given feature.

****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.**