Hunter Leise

CSCI 3202

Problem Set 3

**Problem 3.1**

(For b, c, d, and e, black nodes mean the variable is true, and red nodes mean the variable is false. Black leaf nodes mean that path matches the goal criteria, and red leaf nodes mean that path does not match the goal criteria. The variables are located on the right side of the graph.)

1. P(N, C, L) =0.7 \* 0.4 \* 0.2 = 0.056

P(~N, C, L) = 0.3 \* 0.4 \* 0.6 = 0.072

P(N, ~C, L) = 0.7 \* 0.6 \* 0.5 = 0.21

P(~N, ~C, L) = 0.3 \* 0.6 \* 0.8 = 0.144

P(L) = 0.056 + 0.072 + 0.21 + 0.144

= **0.482**

P(B) = P(B|L)P(L) + P(B|~L)P(~L)

= 0.9(0.482) + 0.2(0.518)

= **0.5374**

P(M) = P(M|L)P(L) + P(M|~L)P(~L)

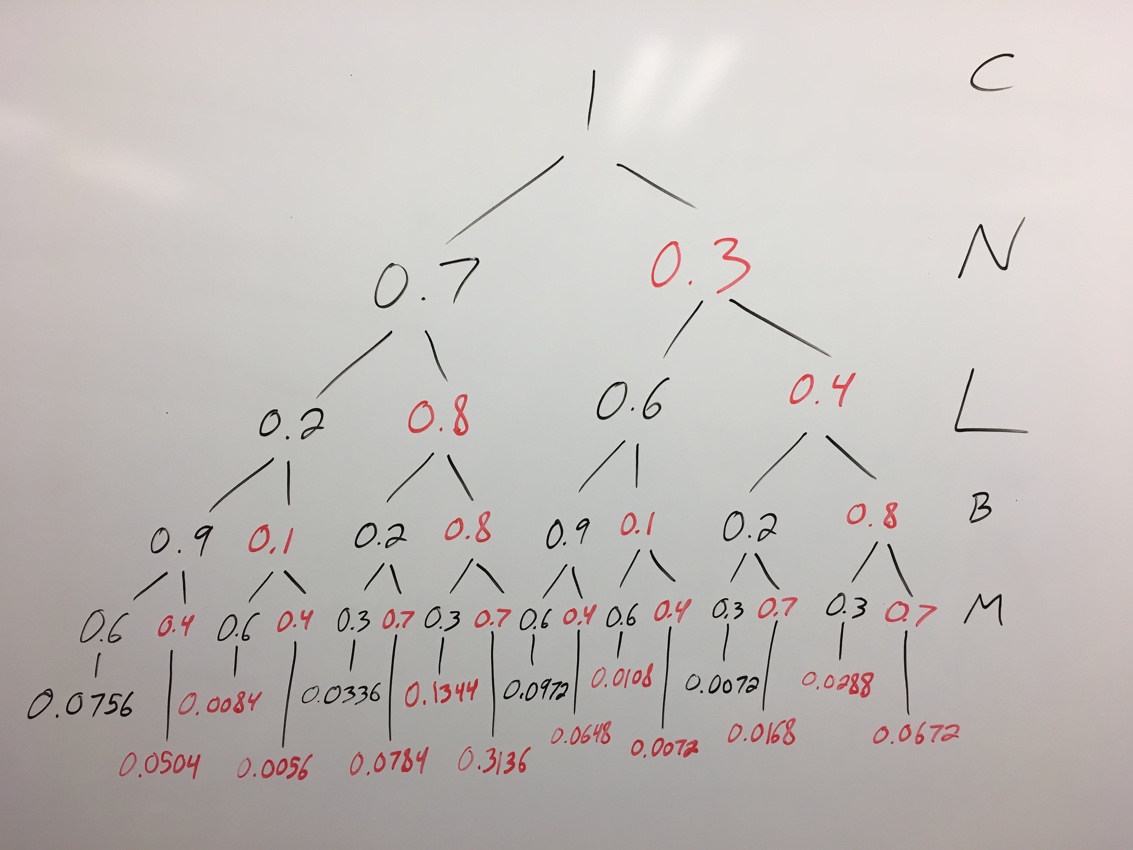
= 0.6(0.482) + 0.3(0.518)

= **0.4446**

P(S) = P(S|M)P(M) + P(S|~M)P(~M)

= 0.8(0.4446) + 0.1(0.5554)

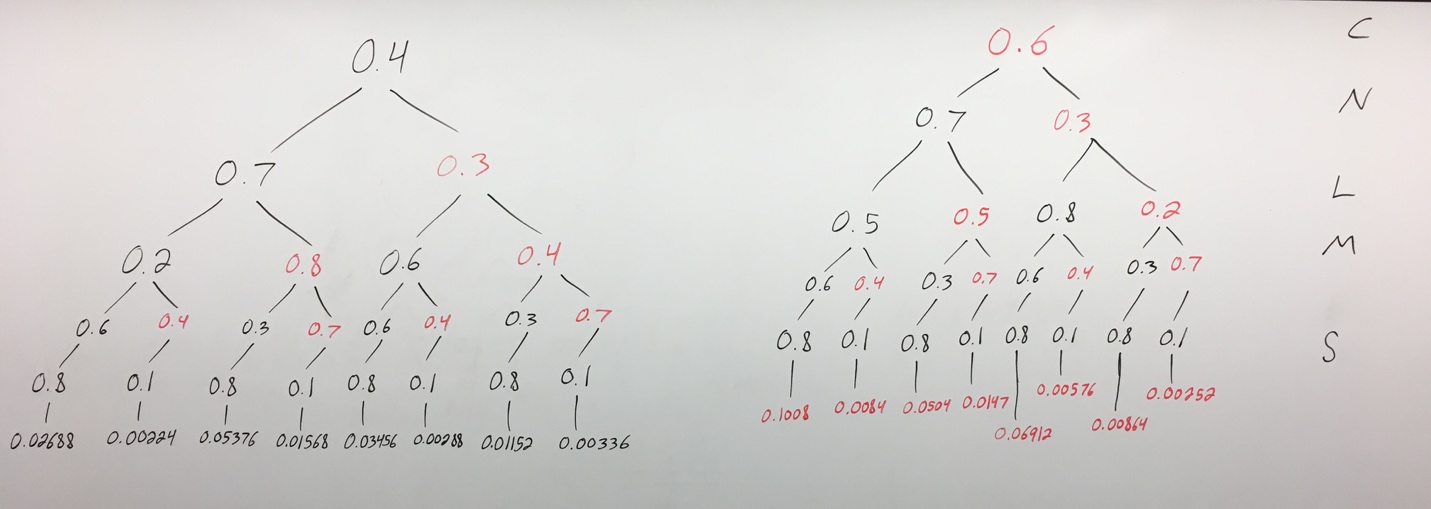
= **0.41122**



P(B, M|C) = summation of black leaf nodes / summation of all leaf nodes

= 0.2136 / 1

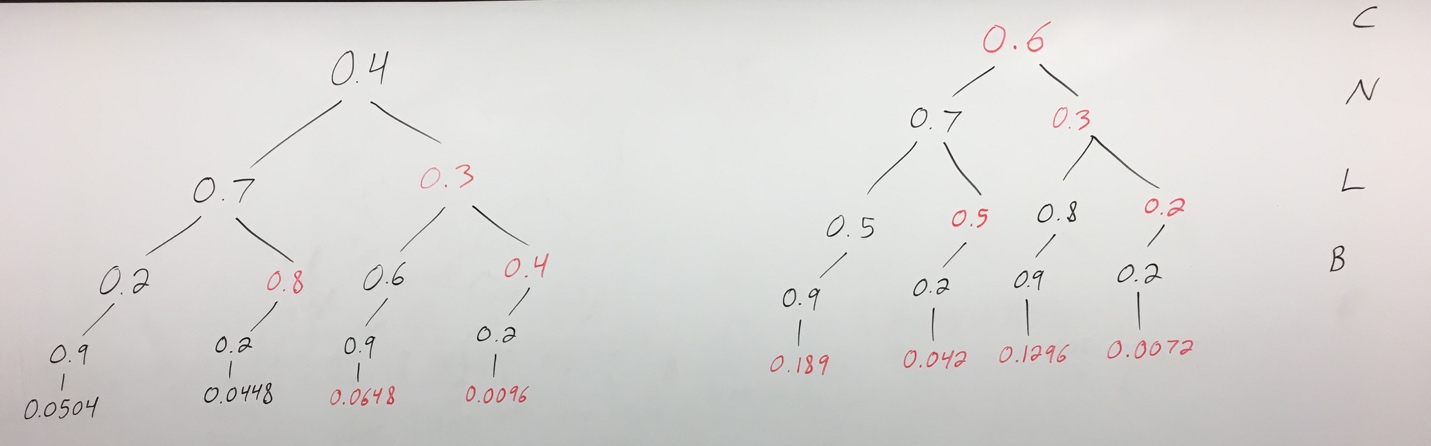
= **0.2136**



P(C|S) = summation of black leaf nodes / summation of all leaf nodes

= 0.15088 / 0.41122

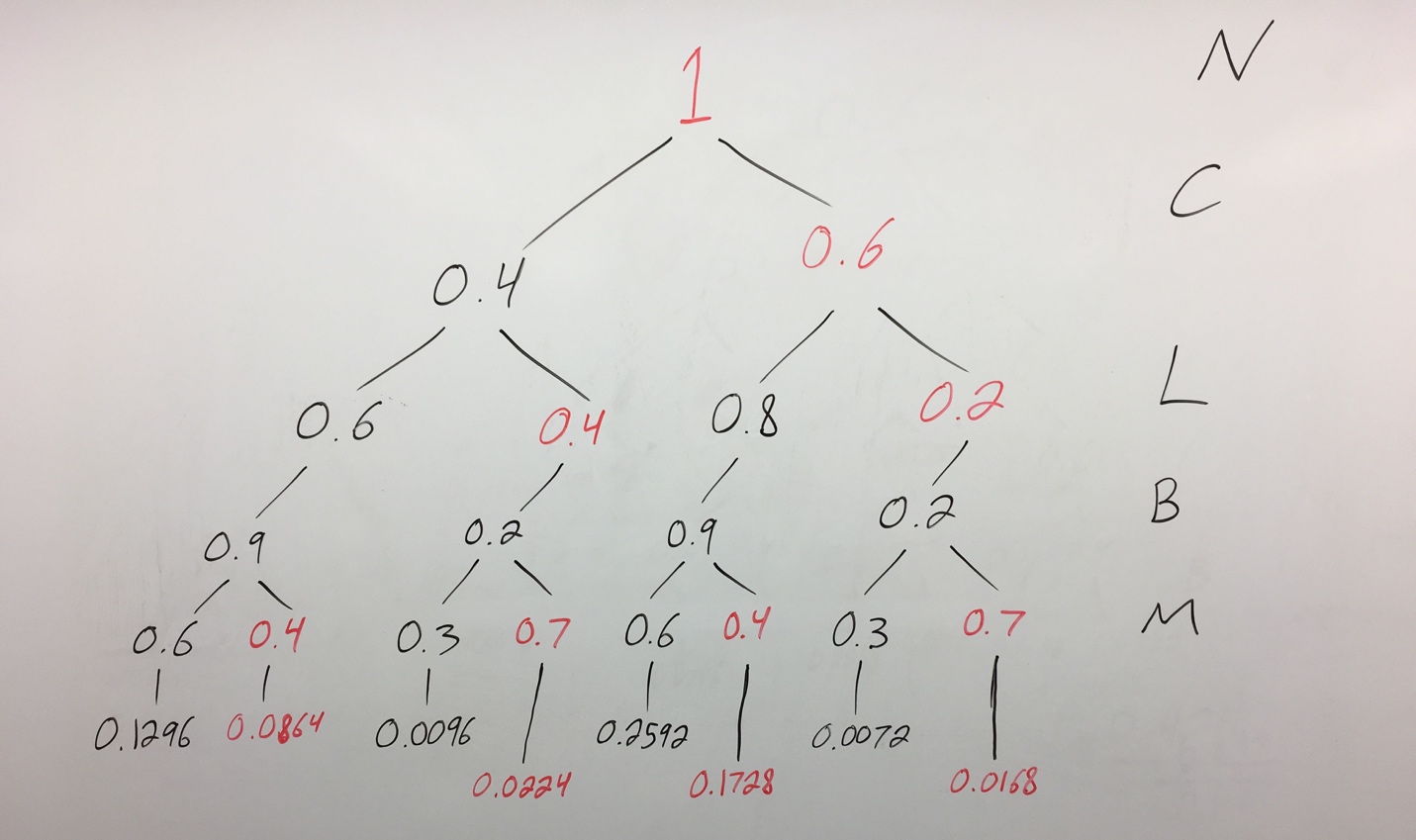
≈ **0.3669**



P(C, N|B) = summation of black leaf nodes / summation of all leaf nodes

= 0.0952 / 0.5374

≈ **0.17715**



P(M|B, ~N) = summation of black leaf nodes / summation of all leaf nodes

= 0.4056 / 0.704

≈ **0.5761**

**Problem 3.2**

1. Starting information =

=

= 1

Attribute A Set 1 =

= 0.6 \* (0.38998 + 0.52832)

= 0.6 \* (0.9183)

≈ 0.55098

Attribute A Set 2 =

= 0.4 \* (0.5 + 0.31128)

= 0.4 \* (0.811280

≈ 0.32451

Attribute A = 0.55098 + 0.324512

≈ 0.87549

Attribute B Set 1 =

= 0.875 \* (0.4613 + 0.5239)

= 0.875 \* (0.9852)

≈ 0.8621

Attribute B Set 2 =

= 0.125 \* (0+ 0)

= 0.125 \* (0)

= 0

Attribute B = 0.8621 + 0

≈ 0.8621

**Attribute B resulted in the most decrease in entropy, so B is the more informative attribute.**

1. Starting information =

=

= 1

Attribute A Set 1 =

= 0.5 \* (0.2575 + 0.4644)

= 0.5 \* (0.7219)

≈ 0.36095

Attribute A Set 2 =

= 0.5 \* (0.4644 + 0.2575)

= 0.5 \* (0.7219)

≈ 0.36095

Attribute A = 0.36095 + 0.36095

≈ 0.7219

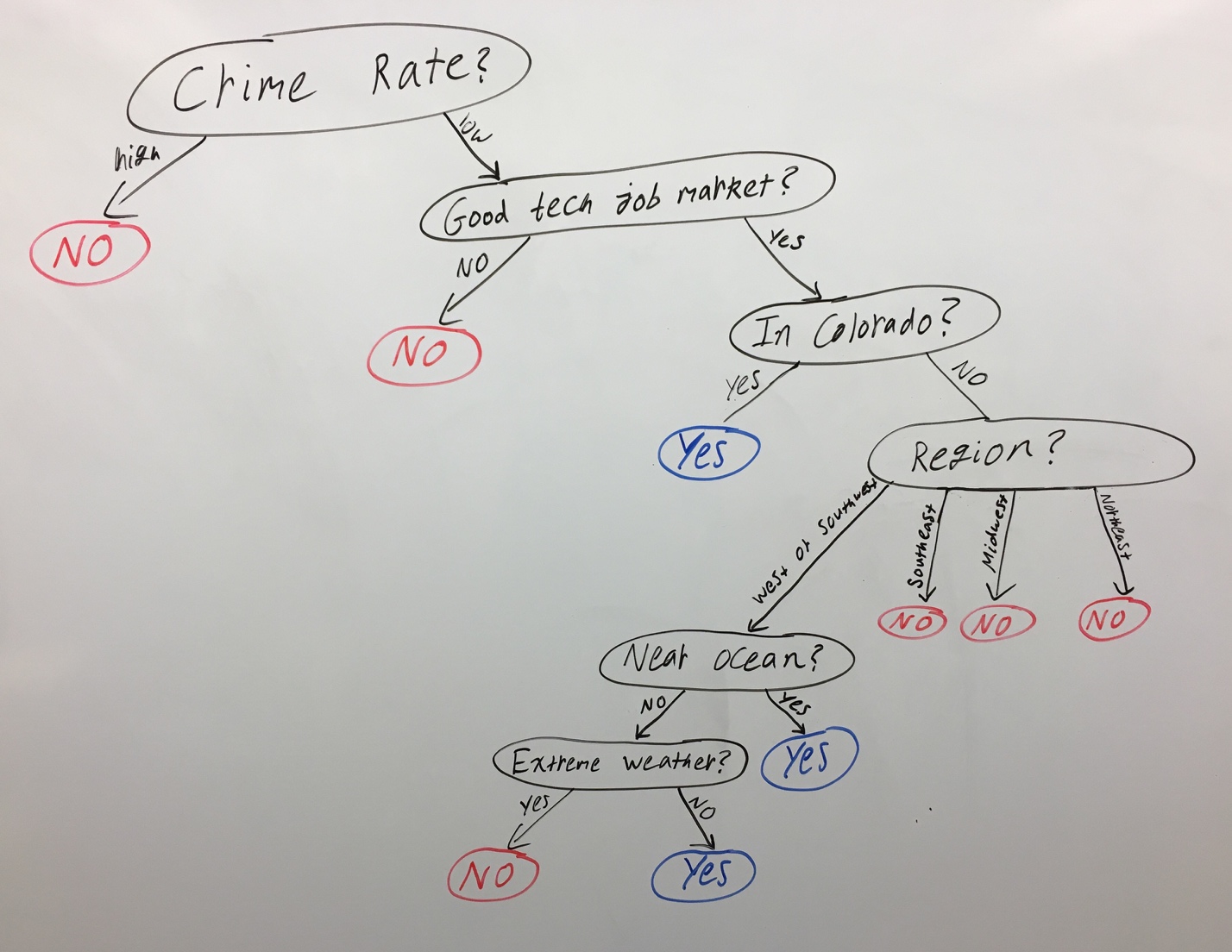
**The largest value of X such that Attribute B is more informative than Attribute A (0.7219) is 27 with an entropy value of ≈0.7197.**

I got this answer by trying different values of x in the following code:

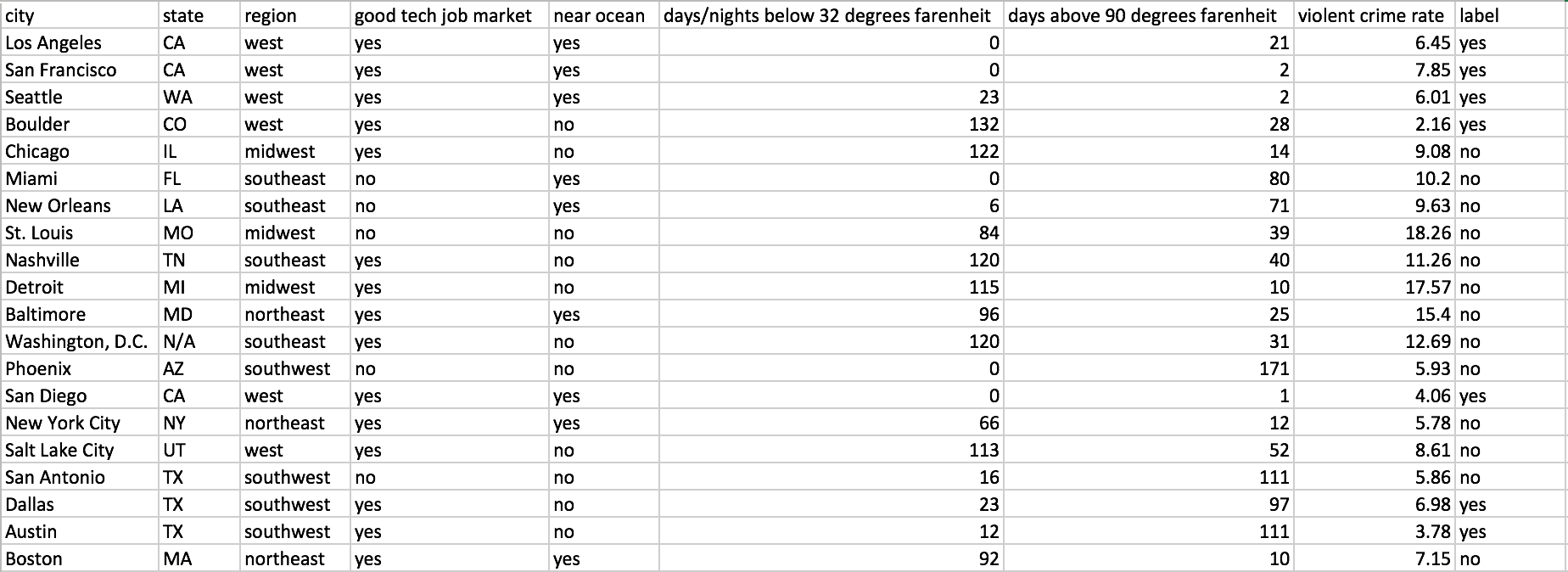
from math import log2  
  
 x = 27  
  
 set1\_yes = x  
 set1\_no = 50  
  
 set2\_yes = x  
 set2\_no = 50  
  
 set3\_yes = (100 - 2 \* x)  
 set3\_no = 0  
  
 set1\_total = set1\_yes + set1\_no  
 set2\_total = set2\_yes + set2\_no  
 set3\_total = set3\_yes + set3\_no  
  
 set1\_weight = set1\_total / (set1\_total + set2\_total + set3\_total)  
 set2\_weight = set2\_total / (set1\_total + set2\_total + set3\_total)  
 set3\_weight = set3\_total / (set1\_total + set2\_total + set3\_total)  
  
 pa = set1\_yes / set1\_total *# P(set 1 yes)* pb = set1\_no / set1\_total *# P(set 1 no)* pc = set2\_yes / set2\_total *# P(set 2 yes* pd = set2\_no / set2\_total *# P(set 2 no)* pe = set3\_yes / set3\_total *# P(set 3 yes)* pf = set3\_yes / set3\_total *# P(set 3 no)* set1\_entropy = set1\_weight \* ((-pa \* log2(pa)) + (-pb \* log2(pb)))  
 set2\_entropy = set2\_weight \* ((-pc \* log2(pc)) + (-pd \* log2(pd)))  
 set3\_entropy = set3\_weight \* ((-pe \* log2(pe)) + (-pf \* log2(pf)))  
  
 entropy = set1\_entropy + set2\_entropy + set3\_entropy  
 print(entropy)

**Problem 3.3**

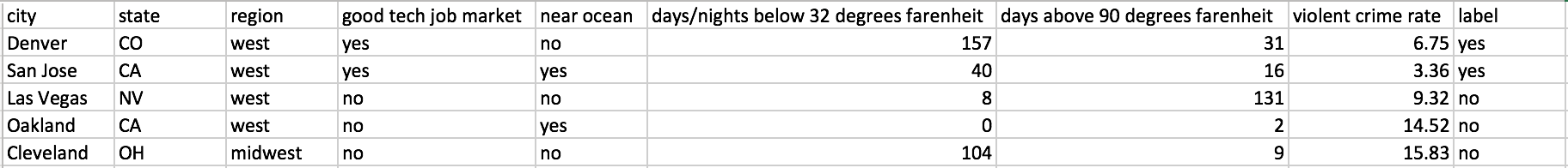
1. Would I like to live here?



1. Training Examples:



Test Examples:

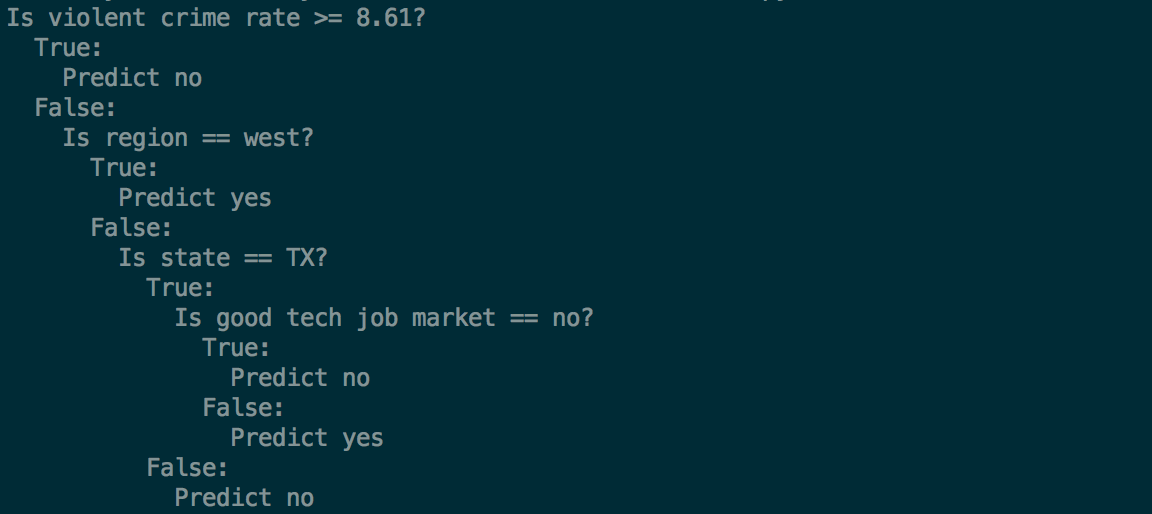


Code:

import csv  
from math import log2  
  
  
*# Returns the headers, cities, and data from the given csv file*def get\_data\_from\_csv(filename):  
 with open(filename, newline=**''**) as csvfile:  
 reader = csv.DictReader(csvfile)  
 headers = reader.fieldnames[1:]  
 cities = []  
 data = []  
 for row in reader:  
 cities.append(row[**'city'**])  
 data.append([row[headers[0]], row[headers[1]], row[headers[2]],  
 row[headers[3]], int(row[headers[4]]),  
 int(row[headers[5]]), float(row[headers[6]]),  
 row[headers[7]]])  
 return headers, cities, data  
  
  
*# Returns the entropy*def entropy(cities, weight):  
 counts = label\_count(cities)  
 total = float(len(cities))  
 p\_yes = counts[**'yes'**] / total  
 p\_no = counts[**'no'**] / total  
 ent\_yes = -p\_yes \* log2(p\_yes) if p\_yes != 0 else 0  
 ent\_no = -p\_no \* log2(p\_no) if p\_no != 0 else 0  
 return weight \* (ent\_yes + ent\_no)  
  
  
*# Returns the information gain from a query*def info\_gain(left, right, current\_uncertainty):  
 weight\_left = float(len(left)) / (len(left) + len(right))  
 weight\_right = float(len(right)) / (len(left) + len(right))  
 total\_entropy = entropy(left, weight\_left) + entropy(right, weight\_right)  
 return current\_uncertainty - total\_entropy  
  
  
*# Leaf node that represents yes or no*class LeafNode:  
 def \_\_init\_\_(self, cities):  
 self.predictions = label\_count(cities)  
  
  
*# Represents a node that splits the data on a question*class QueryNode:  
 def \_\_init\_\_(self, question, true\_branch, false\_branch):  
 self.question = question  
 self.true\_branch = true\_branch  
 self.false\_branch = false\_branch  
  
  
*# Used to query given examples*class Question:  
 def \_\_init\_\_(self, attribute, value):  
 self.attribute = attribute  
 self.value = value  
  
 *# Query if the given example is true for this question* def is\_true(self, example):  
 val = example[self.attribute]  
 if isinstance(val, int) or isinstance(val, float): *# if number* return val >= self.value  
 else: *# if not a number* return val == self.value  
  
 def \_\_str\_\_(self): *# Used to print the question* if isinstance(self.value, int) or isinstance(self.value, float):  
 return **"Is %s >= %s?"** % (headers[self.attribute], str(self.value))  
 else:  
 return **"Is %s == %s?"** % (headers[self.attribute], str(self.value))  
  
  
*# Returns the number of cities with each label*def label\_count(cities):  
 num\_labels = {**'yes'**: 0, **'no'**: 0}  
 for city in cities:  
 label = city[-1] *# label is the last attribute* num\_labels[label] += 1  
 return num\_labels  
  
  
*# Check if the question is true for each given city  
# Returns a list of the true and false cities*def partition(cities, question):  
 true\_cities = []  
 false\_cities = []  
 for city in cities:  
 if question.is\_true(city):  
 true\_cities.append(city)  
 else:  
 false\_cities.append(city)  
 return true\_cities, false\_cities  
  
  
*# Return the best question and its information gain*def get\_best\_question(cities):  
 best\_question = None  
 best\_gain = 0  
 num\_attributes = len(headers) - 1  
  
 for attribute in range(0, num\_attributes):  
 values = set([city[attribute] for city in cities]) *# unique values* for val in values:  
 question = Question(attribute, val)  
  
 *# split data* true\_rows, false\_rows = partition(cities, question)  
 if len(true\_rows) == 0 or len(false\_rows) == 0:  
 continue *# skip if data doesn't split*

*# get info gain for this attribute* current\_uncertainty = entropy(cities, 1)  
 gain = info\_gain(true\_rows, false\_rows, current\_uncertainty)  
 if gain >= best\_gain:  
 best\_gain, best\_question = gain, question  
  
 return best\_question, best\_gain  
  
  
*# Recursively builds the decision tree*def build\_tree(cities):  
 question, gain = get\_best\_question(cities)  
  
 if gain == 0: *# means it must be a leaf node* return LeafNode(cities)  
  
 *# Recursively builds child branches* true\_cities, false\_cities = partition(cities, question)  
 true\_branch = build\_tree(true\_cities)  
 false\_branch = build\_tree(false\_cities)  
  
 return QueryNode(question, true\_branch, false\_branch)  
  
  
*# Recursively print the decision tree*def print\_tree(node, spacing=**""**):  
 *# Base case: leaf node* if isinstance(node, LeafNode):  
 prediction = max(node.predictions, key=node.predictions.get)  
 print(spacing + **"Predict %s"** % prediction)  
 return  
  
 *# Print question* print(spacing + str(node.question))  
  
 *# True branch* print(spacing + **' True:'**)  
 print\_tree(node.true\_branch, spacing + **" "**)  
  
 *# False branch* print(spacing + **' False:'**)  
 print\_tree(node.false\_branch, spacing + **" "**)  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 *# Get data from training csv file* headers, training\_cities, training\_data = \  
 get\_data\_from\_csv(**'trainingData.csv'**)  
  
 *# Use the classifier to build a tree* tree = build\_tree(training\_data)  
 print\_tree(tree)

Computer Generated Tree:



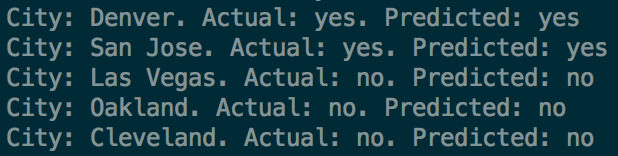
1. The tree produced by the program recognizes that the crime rate is the most useful attribute for splitting the data. It’s actually much more concise than the tree I made in part A by not even needing to use the temperature and ocean attributes. I find it really interesting that it used the crime rate of Salt Lake City as the border between labels. This results in the correct categorization, but the crime rate isn’t the reason I wouldn’t want to live in Salt Lake City.

I definitely see my own decision process in the computer-produced tree. Of course, there are so many factors to why I would choose to live somewhere, but given the few attributes that I listed, the computer did a really good job!

Test code added to above code:

def classify(city, node):  
 *# Base case: leaf node* if isinstance(node, LeafNode):  
 return node.predictions  
  
 *# General case: query node* if node.question.is\_true(city):  
 return classify(city, node.true\_branch)  
 else:  
 return classify(city, node.false\_branch)  
  
  
*# Returns the prediction for a given classification  
# Note that if two values have the same probability,  
# it will only return the first*def get\_prediction(counts):  
 return max(counts, key=counts.get)  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 *# Get data from training and test csv files* headers, training\_cities, training\_data = \  
 get\_data\_from\_csv(**'trainingData.csv'**)  
 \_, test\_cities, test\_data = get\_data\_from\_csv(**'testData.csv'**)  
  
 *# Use the classifier to build a tree* tree = build\_tree(training\_data)  
  
 *# Predict the test cases* for i in range(0, len(test\_data)):  
 print(**"City: %s. Actual: %s. Predicted: %s"** %  
 (test\_cities[i], test\_data[i][-1],  
 get\_prediction(classify(test\_data[i], tree))))

Output:



**Both my cognitive tree and the computer-generated tree outputs the correct prediction for all five test cases.**