**main.py:**

import pandas as pd  
import matplotlib.pyplot as plt  
  
import attributes as att  
import decision\_tree as dt  
  
def increment\_dict\_count(dict, key):  
 if key in dict:  
 dict[key] += 1  
 else:  
 dict[key] = 1  
  
def create\_before\_after\_dictionaries(raw\_data):  
 words\_before = dict()  
 words\_after = dict()  
 for line in raw\_data:  
 line = line.strip().lower()  
 if line == '':  
 continue  
  
 tokens = line.split(' ')  
 target\_index = int(tokens[1])  
 words = tokens[2:len(tokens) - 1]  
  
 if target\_index < 0 or target\_index >= len(words):  
 continue  
  
 if target\_index > 0:  
 increment\_dict\_count(words\_before, words[target\_index - 1])  
 else:  
 increment\_dict\_count(words\_before, '<s>')  
  
 if target\_index < len(words) - 1:  
 increment\_dict\_count(words\_after, words[target\_index + 1])  
 else:  
 increment\_dict\_count(words\_after, '</s>')  
  
 #only return top 50 most occurrences  
 words\_before = {k: v for k, v in sorted(words\_before.items(), key=lambda item: item[1], reverse=True)[:50]}  
 words\_after = {k: v for k, v in sorted(words\_after.items(), key=lambda item: item[1], reverse=True)[:50]}  
 return words\_before, words\_after  
  
def create\_attributes\_data(raw\_data, attributes):  
 formatted\_data = []  
 for line in raw\_data:  
 line = line.strip().lower()  
 if line == '':  
 continue  
 tokens = line.split(' ')  
 true\_label = tokens[0]  
 target\_index = int(tokens[1])  
 words = tokens[2:len(tokens) - 1]  
 if target\_index < 0 or target\_index >= len(words):  
 continue  
 row = [str(attributes[attribute].calculate\_value(true\_label, target\_index, words)) for attribute in attributes]  
 row.append(true\_label)  
 formatted\_data.append(row)  
 return formatted\_data  
  
def create\_test\_cases(raw\_data, attributes):  
 test\_cases = []  
 for line in raw\_data:  
 line = line.strip().lower()  
 if line == '':  
 continue  
 tokens = line.split(' ')  
 true\_label = tokens[0]  
 target\_index = int(tokens[1])  
 words = tokens[2:len(tokens) - 1]  
 if target\_index < 0 or target\_index >= len(words):  
 continue  
 test\_case = {attribute : str(attributes[attribute].calculate\_value(true\_label, target\_index, words)) for attribute in attributes}  
 test\_case['Label'] = true\_label  
 test\_cases.append(test\_case)  
 return test\_cases  
  
def get\_random\_subset(df, num\_examples):  
 return df.sample(num\_examples, ignore\_index=True)  
  
  
### read files  
TRAIN\_FILE\_PATH = 'hw1.train.col'  
TEST\_FILE\_PATH = 'hw1.test.col' # change to 'hw1.dev.col' to test using dev data  
train\_file = open(TRAIN\_FILE\_PATH, 'r', encoding='utf-8')  
test\_file = open(TEST\_FILE\_PATH, 'r', encoding='utf-8')  
train\_data = train\_file.read().split('\n')  
test\_data = test\_file.read().split('\n')  
train\_file.close()  
test\_file.close()  
  
### create dictionary with mapping to word frequency  
words\_before, words\_after = create\_before\_after\_dictionaries(train\_data)  
  
### create attributes/features  
attributes = dict()  
# Add bag of words attributes for words before and after the label word  
for word in words\_before:  
 word\_count = words\_before[word]  
 attribute\_name = '"' + word + '"\_Before'  
 attributes[attribute\_name] = att.WordExistsBeforeAttribute(word)  
for word in words\_after:  
 word\_count = words\_after[word]  
 attribute\_name = '"' + word + '"\_After'  
 attributes[attribute\_name] = att.WordExistsAfterAttribute(word)  
  
  
### create attributes table  
data = create\_attributes\_data(train\_data, attributes)  
column\_names = [attribute for attribute in attributes]  
column\_names.append('Label')  
df = pd.DataFrame(data=data, columns=column\_names)  
  
### create test cases  
test\_cases = create\_test\_cases(test\_data, attributes)  
  
### driver code to test accuracy of decision tree  
# change these 3 variables if needed  
max\_depth = 10  
num\_trials = 3  
sample\_percentage = [0.1, 0.2, 0.5, 0.8, 1]  
  
result\_pairs = []  
divider\_str = ''.rjust(50, '=')  
for sample\_percentage in sample\_percentage:  
 print(divider\_str)  
 accuracy\_avg = 0  
 num\_train\_examples = int(len(df.index) \* sample\_percentage)  
 print('SAMPLE SIZE:', num\_train\_examples, '(' + str(sample\_percentage \* 100) + '% of full training data set)')  
 for i in range(num\_trials):  
 train\_subset = get\_random\_subset(df, num\_train\_examples)  
 print('Building decision tree with max depth ' + str(max\_depth) + '...')  
  
 decision\_tree = dt.DecisionTree(train\_subset, max\_depth)  
 decision\_tree.build\_tree()  
  
 print('GENERATED DECISION TREE:')  
 print(decision\_tree)  
  
 trial\_accuracy = dt.test\_accuracy(decision\_tree, test\_cases)  
 accuracy\_avg += trial\_accuracy  
 print("Accuracy for trial #" + str(i + 1) + ": " + str(round(trial\_accuracy \* 100, 2)) + ".")  
 accuracy\_avg /= num\_trials  
 accuracy\_percentage = round(accuracy\_avg \* 100, 2)  
 result\_pairs.append([num\_train\_examples, accuracy\_percentage])  
 print('\nAVERAGE ACCURACY: ' + str(accuracy\_percentage) + '% (Averaged over ' + str(num\_trials) + ' trials)')  
  
print(divider\_str)  
print(divider\_str)  
print("RESULT:")  
result\_table = pd.DataFrame(data=result\_pairs, columns=['# Train Examples', 'Avg Accuracy'])  
print(result\_table)  
result\_table.plot(x='# Train Examples', y='Avg Accuracy')  
plt.title('Performance on ID3 DT with Max Depth of ' + str(max\_depth))  
plt.show()

**decision\_tree.py:**

import math  
  
  
# used for calculation of entropy  
# without having to worry about values <= 0  
def safe\_log2(value):  
 return 0 if value <= 0 else math.log(value, 2)  
  
  
def calculate\_entropy(df):  
 entropy = 0  
 unique\_labels = df['Label'].unique()  
 for label in unique\_labels:  
 relative\_freq = df['Label'].value\_counts()[label] / len(df['Label'])  
 entropy += -relative\_freq \* safe\_log2(relative\_freq)  
 return entropy  
  
  
def calculate\_normalized\_entropy\_for\_attribute(df, attribute):  
 unique\_labels = df['Label'].unique()  
 unique\_value\_for\_att = df[attribute].unique()  
 entropy\_normalized = 0  
 for value in unique\_value\_for\_att:  
 entropy = 0  
 for label in unique\_labels:  
 num\_value\_occurrences\_for\_label = len(df[(df['Label'] == label) & (df[attribute] == value)])  
 relative\_freq = num\_value\_occurrences\_for\_label / df[attribute].value\_counts()[value]  
 entropy += -relative\_freq \* safe\_log2(relative\_freq)  
 num\_value\_occurrences = len(df.loc[df[attribute] == value])  
 entropy\_normalized += entropy \* (num\_value\_occurrences / len(df[attribute]))  
 return entropy\_normalized  
  
  
def calculate\_info\_gain\_for\_attribute(df, attribute):  
 return calculate\_entropy(df) - calculate\_normalized\_entropy\_for\_attribute(df, attribute)  
  
  
def get\_max\_info\_gain\_attribute(df):  
 data\_only\_df = df.drop('Label', axis=1)  
 max\_info\_gain = -1  
 max\_info\_gain\_attribute = None  
 for attribute in data\_only\_df.columns:  
 current\_att\_info\_gain = calculate\_info\_gain\_for\_attribute(df, attribute)  
 if current\_att\_info\_gain > max\_info\_gain:  
 max\_info\_gain = current\_att\_info\_gain  
 max\_info\_gain\_attribute = attribute  
 if max\_info\_gain <= 0:  
 return None  
 return max\_info\_gain\_attribute  
  
  
def get\_subtable\_for\_attribute\_value(df, attribute, value):  
 return df[df[attribute] == value].reset\_index(drop=True)  
  
  
def subtable\_is\_homogenous(df):  
 return len(df['Label'].unique()) == 1  
  
  
def test\_accuracy(decision\_tree, test\_cases):  
 correct\_counter = 0  
 for test\_case in test\_cases:  
 predicted\_label = decision\_tree.decide(test\_case)  
 if predicted\_label == test\_case['Label']:  
 correct\_counter += 1  
 return correct\_counter / len(test\_cases)  
  
  
class Node:  
 def \_\_init\_\_(self, subtable, attribute):  
 self.subtable = subtable  
 self.attribute = attribute  
 self.children = []  
  
 def get\_label(self):  
 return self.subtable['Label'].value\_counts().idxmax()  
  
 def \_\_repr\_\_(self):  
 if self.attribute is None:  
 return 'Root'  
 s = self.attribute + ': ' + self.subtable[self.attribute][0]  
 if len(self.children) == 0:  
 value\_counts = self.subtable['Label'].value\_counts()  
 max\_label = value\_counts.idxmax()  
 percentage\_of\_occurrence = round((value\_counts[max\_label] / len(self.subtable['Label'])) \* 100, 2)  
 s += (' --> ' + max\_label + ' (' + str(percentage\_of\_occurrence) + '%)')  
 return s  
  
  
class DecisionTree:  
  
 def \_\_init\_\_(self, df, max\_depth):  
 self.root = Node(df, None)  
 self.max\_depth = max\_depth  
  
 def build\_tree(self):  
 stack = [{'current\_node': self.root, 'current\_level': 0}]  
 while len(stack) > 0:  
 current\_level = stack[-1]['current\_level']  
 current\_node = stack[-1]['current\_node']  
 stack.pop()  
  
 if current\_level >= self.max\_depth:  
 continue  
  
 attribute\_to\_split\_on = get\_max\_info\_gain\_attribute(current\_node.subtable)  
 if attribute\_to\_split\_on is None:  
 continue  
  
 unique\_values\_for\_attribute = current\_node.subtable[attribute\_to\_split\_on].unique()  
 for value in unique\_values\_for\_attribute:  
 new\_node\_subtable = get\_subtable\_for\_attribute\_value(current\_node.subtable, attribute\_to\_split\_on, value)  
 new\_node = Node(new\_node\_subtable, attribute\_to\_split\_on)  
 current\_node.children.append(new\_node)  
 if not subtable\_is\_homogenous(new\_node\_subtable):  
 stack.append({'current\_node': new\_node, 'current\_level': current\_level + 1})  
  
 def decide(self, test\_case):  
 stack = [self.root]  
 while len(stack) > 0:  
 current\_node = stack.pop()  
 if len(current\_node.children) == 0:  
 return current\_node.get\_label()  
 for child in current\_node.children:  
 if child.subtable[child.attribute][0] == test\_case[child.attribute]:  
 stack.append(child)  
  
 def \_\_repr\_\_(self):  
 s = ""  
 stack = [{'current\_node': self.root, 'current\_level': 0}]  
 while len(stack) > 0:  
 current\_level = stack[-1]['current\_level']  
 current\_node = stack[-1]['current\_node']  
 stack.pop()  
  
 if current\_node is None:  
 continue  
  
 padding\_str = ''.rjust(current\_level, '\t')  
 node\_str = current\_node.\_\_repr\_\_()  
 s += (padding\_str + node\_str + '\n')  
  
 for child in current\_node.children:  
 stack.append({'current\_node': child, 'current\_level': current\_level + 1})  
 return s

**attributes.py:**

# This is an interface  
class Attribute:  
 # This is an abstract method and is meant to be overriden by inheriting classes  
 def calculate\_value(self, true\_label, target\_index, words):  
 pass  
  
  
class WordExistsBeforeAttribute(Attribute):  
 def \_\_init\_\_(self, word):  
 super().\_\_init\_\_()  
 self.word = word  
  
 def calculate\_value(self, true\_label, target\_index, words):  
 return target\_index > 0 and words[target\_index - 1] == self.word  
  
  
class WordExistsAfterAttribute(Attribute):  
 def \_\_init\_\_(self, word):  
 super().\_\_init\_\_()  
 self.word = word  
  
 def calculate\_value(self, true\_label, target\_index, words):  
 return target\_index < len(words) - 1 and words[target\_index + 1] == self.word