CS189 HW5 Yong-Chan Shin

1. Decision Tree Code

```
class Node:
        def __init__(self,data,labels):
                self.data = data
                self.labels = labels
                self.left = np.asarray([])
                self.right = np.asarray([])
                self.label = 0
                self.split_rule = (0,0) #(feature_index,threshold)
                self.isLeaf = False


class decisionTree:
        def __init__(self,max_depth,num_random_feature_subset):
                self.max_depth = max_depth
                self.num_feature_subset = num_random_feature_subset
                self.feature_index_subset = []
        def impurity(self,left_label_hist,right_label_hist):
                left_entropy = 0
                right_entropy = 0
                left_total_cardinal = 0
                right_total_cardinal = 0
                for a_class, class_cardinal in left_label_hist.items():
                        #print("insider of impurity/firstiterleftnodecount")
                        #print("a_class:",a_class," cardinal:",class_cardinal)
                        left_total_cardinal+=class_cardinal
                for a_class, class_cardinal in right_label_hist.items():
                        right_total_cardinal+=class_cardinal
                if left_total_cardinal != 0:
                        for a_class, class_cardinal in left_label_hist.items():
                                p_c = float(class_cardinal)/left_total_cardinal
                                if p_c!=0:
```

```
                                        left_entropy -= p_c*log(p_c,2)
                if right_total_cardinal != 0:
                        for a_class, class_cardinal in right_label_hist.items():
                                p_c = float(class_cardinal)/right_total_cardinal
                                if p_c!=0:
                                        right_entropy -= p_c*log(p_c,2)
#               print("l card",left_total_cardinal)
#               print("r card",right_total_cardinal)
#               print("l ent",left_entropy)
#               print("r ent",right_entropy)
                return
float(left_total_cardinal*left_entropy+right_total_cardinal*right_entropy)/(left_total_cardinal+right_total_cardinal)


        def segmenter(self,data,labels):
                #print("inside of segmenter")
                best_feature_index = 0 #default
                best_feature_val = -1 #default
                best_entropy = float("inf") #default
                best_left_node = []
                best_right_node = []
                best_left_node_label = []
                best_right_node_label = []
                for feature_index in self.feature_index_subset: #change here for random forest
range(data.shape[1])
                        #print("curr feature_index",feature_index)
                        #feature_data = np.concatenate((data,np.asarray([[label]for label in labels])),axis=1)
                        #print("labels:",labels)
                        #print("dataT:",data.T)
                        feature_data = np.concatenate((data.T,[labels]),axis=0).T
                        #print("feature_data shape:",feature_data.shape)
                        feature_data = np.asarray(sorted(feature_data.tolist(),key=lambda x:x[feature_index]))
                        #print("sorted features after sort:",feature_data[:,feature_index])
                        feature_list = feature_data[:,:feature_data.shape[1]-1]
```

```
                feature_labels = feature_data[:,feature_data.shape[1]-1]
                #print("number of zero in this feature:",len([i for i in feature_list[:,feature_index] if i==0]))
                feature_val_index = 0
                label_classes, counts = np.unique(labels,return_counts=True)
                left_node_hist = {label_class:0 for label_class in label_classes}
                right_node_hist = dict(zip(label_classes, counts))
                while feature_val_index < feature_list.shape[0]:
                    #print("curr feature_val_index",feature_val_index)
                    curr_index = feature_val_index
                    #print("b4 iter left_node_hist",left_node_hist)
                    #print("b4 iter right_node_hist",right_node_hist)
                    while feature_val_index<feature_list.shape[0] and feature_list[feature_val_index,feature_index]==feature_list[curr_index,feature_index]:
                        left_node_hist[feature_labels[feature_val_index]]+=1
                        right_node_hist[feature_labels[feature_val_index]]-=1
                        #print("curr sample index",feature_val_index)
                        #print("left node hist")
                        #print(left_node_hist)
                        #print("right node hist")
                        #print(right_node_hist)
                        #print("")
                        feature_val_index+=1
                    split_point = feature_list[curr_index,feature_index]
                    H_after = self.impurity(left_node_hist,right_node_hist)
                    #print("curr entropy",H_after)
                    if H_after < best_entropy: #minimize H_after
                        #print("Entropy renewed")
                        #print("Renewed feature_index",feature_index)
                        best_feature_index = feature_index
                        best_entropy = H_after
                        best_feature_val = split_point
                        best_left_node = feature_list[:feature_val_index,:]
```

```
                                        best_left_node_label = feature_labels[:feature_val_index]
                                        best_right_node = feature_list[feature_val_index:,:]
                                        best_right_node_label = feature_labels[feature_val_index:]
                return best_feature_index, best_feature_val, best_left_node, best_right_node, best_left_node_label, best_right_node_label


        def recursive_segmenter(self,node,depth):
                #print("curr depth:",self.max_depth-depth)
                mode_result = mode(node.labels)
                node.label = mode_result[0][0]
                label_classes,counts = np.unique(node.labels,return_counts=True)
                freq_result = dict(zip(label_classes, counts))
                #print("freq_result:",freq_result)
                #print("mode_result:",mode_result)
                if depth!=0 and mode_result[1][0]!=len(node.labels) and float(mode_result[1][0])/len(node.labels) < 0.9:
                        best_feature_index, best_feature_val, best_left_node, best_right_node, best_left_node_label, best_right_node_label = self.segmenter(node.data,node.labels)
                        if len(best_left_node)==0 or len(best_right_node)==0:
                                node.isLeaf = True
                        else:
                                node.split_rule = (best_feature_index,best_feature_val)
                                node.left = Node(best_left_node,best_left_node_label)
                                node.right = Node(best_right_node,best_right_node_label)
                                #print("curr node left length vs right length:",len(best_left_node),"vs",len(best_right_node))
                                #print("split_rule(best feature index, best feature val) got:",node.split_rule)
                                self.recursive_segmenter(node.left,depth-1)
                                self.recursive_segmenter(node.right,depth-1)
                else:
                        node.isLeaf = True
        def train(self,data,labels):
                self.root = Node(data,labels)
                self.feature_index_subset = np.random.permutation(data.shape[1])[:self.num_feature_subset]
```

```
                #print("feature_index_subset",self.feature_index_subset)
                #self.feature_index_subset = np.random.randint(data.shape[1],size=self.num_random_feature_subset)
                self.recursive_segmenter(self.root,self.max_depth)

        def recursive_predict(self,sample,node):
                feature_index, threshold = node.split_rule
                if node.isLeaf:
                        return node.label
                else:
                        if sample[feature_index]<=threshold:
                                return self.recursive_predict(sample,node.left)
                        else:
                                return self.recursive_predict(sample,node.right)
        def predict(self,data):
                pred_list = []
                for sample in data:
                        prediction = self.recursive_predict(sample,self.root)
                        pred_list.append(int(float(prediction)))
                return pred_list
```

2. Random Forest

```
class randomForest:
    def __init__(self,max_depth,num_random_feature_subset,num_trees):
        self.max_depth = max_depth
        self.num_random_feature_subset = num_random_feature_subset
        self.num_trees = num_trees
        self.tree_list = []
    def train(self,data,labels):
        for tree in range(self.num_trees):
            data,labels = shuffle(data,labels)
            tree = decisionTree(self.max_depth,self.num_random_feature_subset)
            tree.train(data[data.shape[0]//5:,:],labels[data.shape[0]//5:])
            self.tree_list.append(tree)
    def predict(self,data):
        pred_list = []
        for tree in self.tree_list:
            a_pred = tree.predict(data)
            pred_list.append(a_pred)
        pred_list = np.asarray(pred_list)
        final_pred = []
        for pred_index in range(pred_list.shape[1]):
            sample_pred = mode(pred_list[:,pred_index])[0][0]
            final_pred.append(sample_pred)
        return final_pred
```

3(a): If the feature is categorical, make every feature value be a new feature and let 1 if the sample has the feature value and 0 if not. Drop the original categorical feature after appending the new features. If there is missing values, replace it with mode value.

3(b): Let the node be leaf and stop recursion if 1. the recursion reaches the maximum depth, 2. mode value takes 90% of the node's samples (if a single feature value dominates the node).

3(c): I initially stopped recursion if a feature value 100% dominates the node. However, to speed up, I change the criteria to 90%. Also, I dropped highly unrelated categorical values to speed up. Titanic with a single decision tree takes a second to run.

3(d): Take square root number of features for the features to use for each tree, and take 80% of the sample of the total training set with replacement. Create multiple trees and takes mode for each feature from the trees' predictions.

4.DT denotes a single decision tree, RF denotes random forest, t score denotes training set score, and v score denotes validation set score

```
spam DT t score: 0.8329817529796435
spam DT v score: 0.7987341772151899
spam RF t score: 0.7765003691593714
spam RF v score: 0.769409282700422
census DT t_score: 0.8495034377387318
census DT v_score: 0.8534535452322738
census RF t_score: 0.7908708938120703
census RF v_score: 0.780562347188264
titanic DT t_score: 0.89125
titanic DT v_score: 0.7386934673366834
titanic RF t_score: 0.76875
titanic RF v_score: 0.7638190954773869
```

Kaggle display name: YONG-CHAN_SHIN

Spam score: 0.81290

Census score: 0.84949

Titanic score: 0.80860

5(a): Only used the pre-implemented features.

5(b):

Ham example: (feature index 28 is “exclamation”, feature index 29 is “para”, feature index 3 is “money”, feature index 9 is “featured”, feature index 13 is “other”)

```
(i) feature_index 28
is less than or equal to threshold 0.0
(ii)feature_index 29
is larger than threshold 0.0
(iii)feature_index 3
is less than or equal to threshold 0.0
(iv)feature_index 9
is less than or equal to threshold 0.0
(v)feature_index 13
is larger than threshold 0.0
therefore the result is 0.0
```

Spam example: (feature index 28 is “exclamation”, feature index 29 is “para”, feature index 3 is “money”, feature index 9 is “featured”)

```
(i)feature_index 28
is less than or equal to threshold 0.0
(ii)feature_index 29
is larger than threshold 0.0
(iii)feature_index 3
is less than or equal to threshold 0.0
(iv)feature_index 9
is larger than threshold 0.0
therefore the result is 1.0
```

5(c): Using the first sample of the data set and 40 trees

(i) feature index 28 > 0 (9 trees)

(ii) feature index 6 <= 0 (8 trees)

(iii) feature index 29 > 0 (5 trees)

6(a): Transformed the categorical features as stated in 3(a)

6(b): (feature index 54 is “Married-civ-spouse”, feature index 102 is “education-num”, feature index 100 is “capital-gain”, and feature index 101 is “capital-loss”)

Prediction 0:

```
feature_index 54
is less than or equal to threshold 0
therefore the result is 0
```

Prediction 1:

```
feature_index 54
is larger than threshold 0
feature_index 102
is less than or equal to threshold 16
feature_index 102
is larger than threshold 12
feature_index 100
is less than or equal to threshold 5013
feature_index 101
is larger than threshold 1740
therefore the result is 1
```

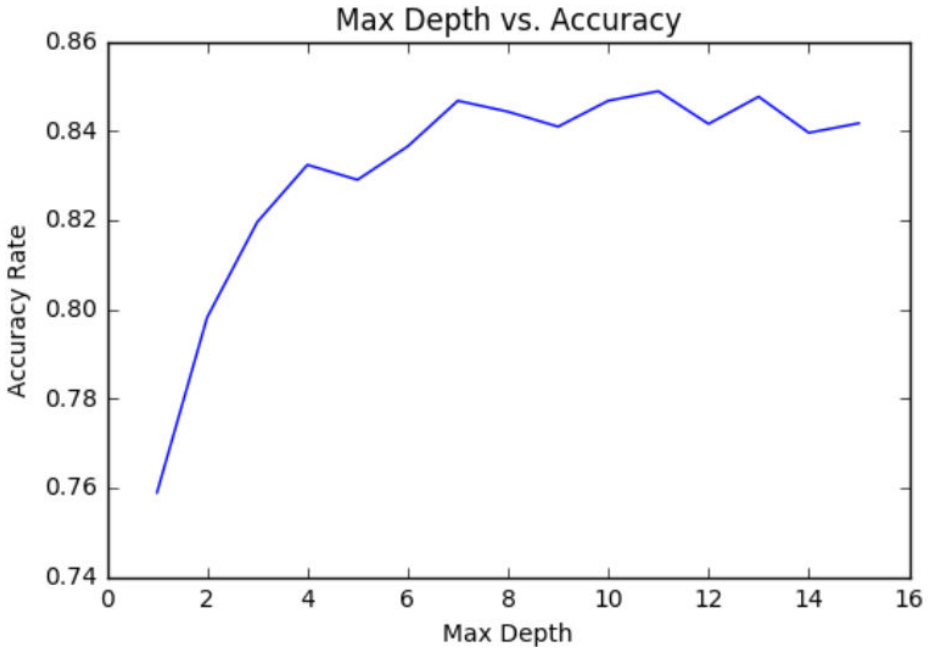
6(c): (feature index 102 is “education-num”, feature index 100 is “capital-gain”, feature index 99 is “age”)

(i) feature_index 102 <= 16 (4 trees)

(ii) feature_index 99 <= 28 (4 trees)

(iii) feature index 100 <= 6849 (3 trees)

6(d):



Additional values that are not on the graph above:

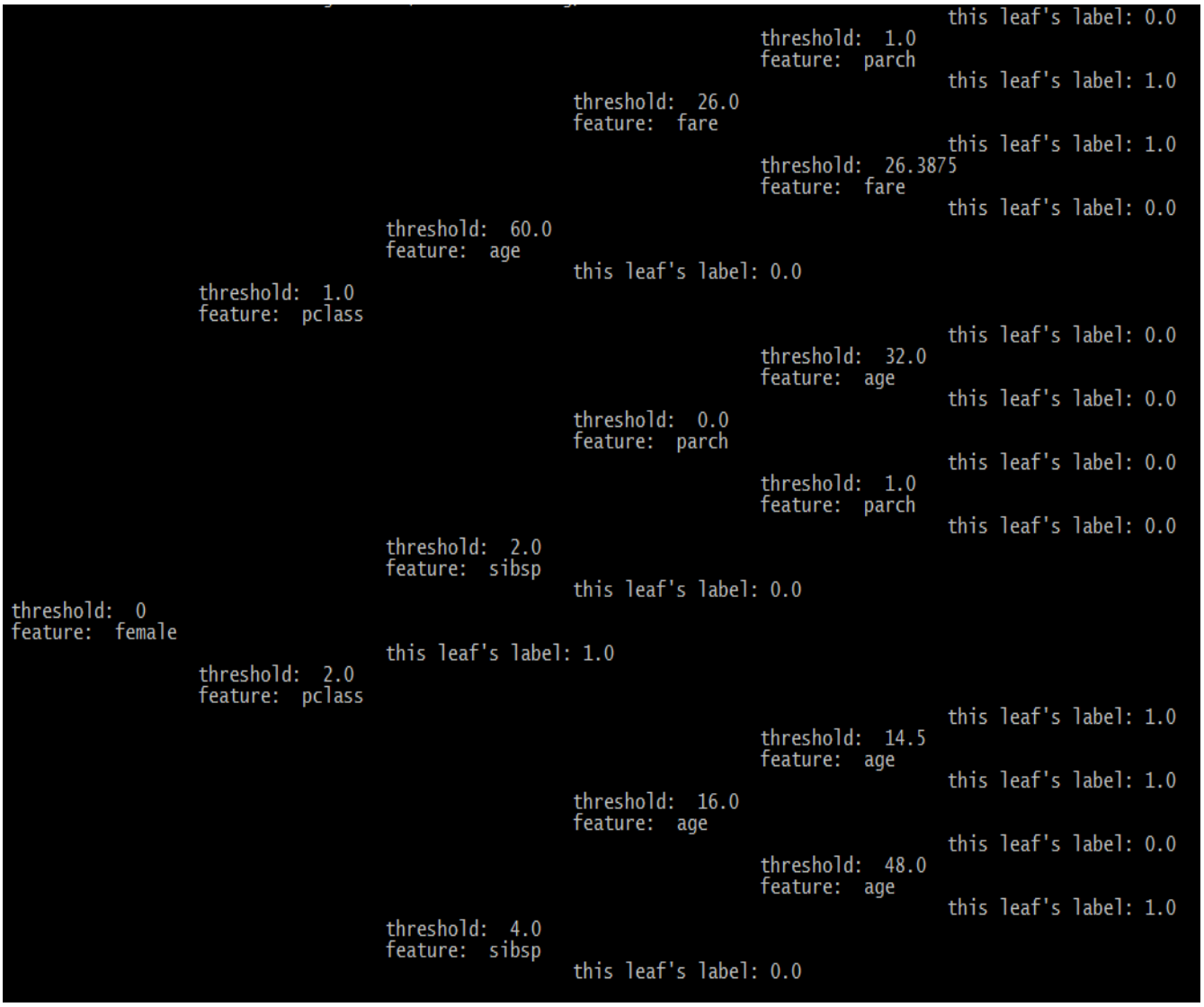
```
current max_depth: 16
```

```
score: 0.8403117359413202

current max_depth: 17
score: 0.8409229828850856

current max_depth: 18
score: 0.840158924205379

current max_depth: 19
score: 0.8404645476772616

current max_depth: 20
score: 0.8384779951100244

current max_depth: 21
score: 0.8390892420537898

current max_depth: 22
score: 0.8381723716381418

current max_depth: 23
score: 0.843062347188264

current max_depth: 24
score: 0.8361858190709046

current max_depth: 25
score: 0.8467298288508558

current max_depth: 26
score: 0.8283924205378973

current max_depth: 27
score: 0.8300733496332519

current max_depth: 28
score: 0.82869804400978

current max_depth: 29
score: 0.8341992665036675

current max_depth: 30
score: 0.8349633251833741
```

Depth 11 gives the highest validation accuracy. The accuracy increases at first, but it does not increase vividly after depth of seven; sometimes it increases, sometimes it decreases slightly.

7(a):

Left-most node is the root node, and as the tree goes right, it is going towards leaves, and the top-most tree is the left-most leaf of the commonly visualized tree. Thus, this tree is a flipped version of a commonly visualized tree.



<CODE>:

```
import csv
import io
from scipy.io import loadmat
from sklearn.feature_extraction import DictVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import Imputer
from math import log
import numpy as np
from scipy.stats import mode
from math import exp
from sklearn.utils import shuffle
from math import sqrt
import pandas as pd
import matplotlib.pyplot as plt

def main():
        #toy1()
        #toy2()
        spam_DT()
        spam_RF()
        census()
        census_RF()
        #q6_d()
        #q7_a()
        titanic()
        titanic_implemented_RF()
        #titanic_kaggle()
        #spam_kaggle()
        #census_kaggle()

def compute_score(labels,pred):
```

```
        error_count = 0
        for tup in zip(pred,labels):
                if tup[0]!=tup[1]:
                        error_count+=1
        return float(len(labels)-error_count)/float(len(labels))


class Node:
        def __init__(self,data,labels):
                self.data = data
                self.labels = labels
                self.left = np.asarray([])
                self.right = np.asarray([])
                self.label = 0
                self.split_rule = (0,0) #(feature_index,threshold)
                self.isLeaf = False


class decisionTree:
        def __init__(self,max_depth,num_random_feature_subset):
                self.max_depth = max_depth
                self.num_feature_subset = num_random_feature_subset
                self.feature_index_subset = []
        def impurity(self,left_label_hist,right_label_hist):
                left_entropy = 0
                right_entropy = 0
                left_total_cardinal = 0
                right_total_cardinal = 0
                for a_class, class_cardinal in left_label_hist.items():
                        #print("insider of impurity/firstiterleftnodecount")
                        #print("a_class:",a_class," cardinal:",class_cardinal)
                        left_total_cardinal+=class_cardinal
                for a_class, class_cardinal in right_label_hist.items():
                        right_total_cardinal+=class_cardinal
```

```
                if left_total_cardinal != 0:
                        for a_class, class_cardinal in left_label_hist.items():
                                p_c = float(class_cardinal)/left_total_cardinal
                                if p_c!=0:
                                        left_entropy -= p_c*log(p_c,2)
                if right_total_cardinal != 0:
                        for a_class, class_cardinal in right_label_hist.items():
                                p_c = float(class_cardinal)/right_total_cardinal
                                if p_c!=0:
                                        right_entropy -= p_c*log(p_c,2)
#               print("l card",left_total_cardinal)
#               print("r card",right_total_cardinal)
#               print("l ent",left_entropy)
#               print("r ent",right_entropy)
                return
float(left_total_cardinal*left_entropy+right_total_cardinal*right_entropy)/(left_total_cardinal+right_total_cardinal)


        def segmenter(self,data,labels):
                #print("inside of segmenter")
                best_feature_index = 0 #default
                best_feature_val = -1 #default
                best_entropy = float("inf") #default
                best_left_node = []
                best_right_node = []
                best_left_node_label = []
                best_right_node_label = []
                for feature_index in self.feature_index_subset: #change here for random forest
range(data.shape[1])
                        #print("curr feature_index",feature_index)
                        #feature_data = np.concatenate((data,np.asarray([[label]for label in labels])),axis=1)
                        #print("labels:",labels)
                        #print("dataT:",data.T)
                        feature_data = np.concatenate((data.T,[labels]),axis=0).T
```

```
                    #print("feature_data shape:",feature_data.shape)
                    feature_data = np.asarray(sorted(feature_data.tolist(),key=lambda x:x[feature_index]))
                    #print("sorted features after sort:",feature_data[:,feature_index])
                    feature_list = feature_data[:,:feature_data.shape[1]-1]
                    feature_labels = feature_data[:,feature_data.shape[1]-1]
                    #print("number of zero in this feature:",len([i for i in feature_list[:,feature_index] if i==0]))
                    feature_val_index = 0
                    label_classes, counts = np.unique(labels,return_counts=True)
                    left_node_hist = {label_class:0 for label_class in label_classes}
                    right_node_hist = dict(zip(label_classes, counts))
                    while feature_val_index < feature_list.shape[0]:
                        #print("curr feature_val_index",feature_val_index)
                        curr_index = feature_val_index
                        #print("b4 iter left_node_hist",left_node_hist)
                        #print("b4 iter right_node_hist",right_node_hist)
                        while feature_val_index<feature_list.shape[0] and feature_list[feature_val_index,feature_index]==feature_list[curr_index,feature_index]:
                            left_node_hist[feature_labels[feature_val_index]]+=1
                            right_node_hist[feature_labels[feature_val_index]]-=1
                            #print("curr sample index",feature_val_index)
                            #print("left node hist")
                            #print(left_node_hist)
                            #print("right node hist")
                            #print(right_node_hist)
                            #print("")
                            feature_val_index+=1
                        split_point = feature_list[curr_index,feature_index]
                        H_after = self.impurity(left_node_hist,right_node_hist)
                        #print("curr entropy",H_after)
                        if H_after < best_entropy: #minimize H_after
                            #print("Entropy renewed")
                            #print("Renewed feature_index",feature_index)
```

```
                                        best_feature_index = feature_index
                                        best_entropy = H_after
                                        best_feature_val = split_point
                                        best_left_node = feature_list[:feature_val_index,:]
                                        best_left_node_label = feature_labels[:feature_val_index]
                                        best_right_node = feature_list[feature_val_index:,:]
                                        best_right_node_label = feature_labels[feature_val_index:]
                return best_feature_index, best_feature_val, best_left_node, best_right_node, best_left_node_label, best_right_node_label


        def recursive_segmenter(self,node,depth):
                #print("curr depth:",self.max_depth-depth)
                mode_result = mode(node.labels)
                node.label = mode_result[0][0]
                label_classes,counts = np.unique(node.labels,return_counts=True)
                freq_result = dict(zip(label_classes, counts))
                #print("freq_result:",freq_result)
                #print("mode_result:",mode_result)
                if depth!=0 and mode_result[1][0]!=len(node.labels) and float(mode_result[1][0])/len(node.labels) < 0.9:
                        best_feature_index, best_feature_val, best_left_node, best_right_node, best_left_node_label, best_right_node_label = self.segmenter(node.data,node.labels)
                        if len(best_left_node)==0 or len(best_right_node)==0:
                                node.isLeaf = True
                        else:
                                node.split_rule = (best_feature_index,best_feature_val)
                                node.left = Node(best_left_node,best_left_node_label)
                                node.right = Node(best_right_node,best_right_node_label)
                                #print("curr node left length vs right length:",len(best_left_node),"vs",len(best_right_node))
                                #print("split_rule(best feature index, best feature val) got:",node.split_rule)
                                self.recursive_segmenter(node.left,depth-1)
                                self.recursive_segmenter(node.right,depth-1)
                else:
```

```
                node.isLeaf = True
        def train(self,data,labels):
                self.root = Node(data,labels)
                self.feature_index_subset = np.random.permutation(data.shape[1])[:self.num_feature_subset]
                #print("feature_index_subset",self.feature_index_subset)
                #self.feature_index_subset =
np.random.randint(data.shape[1],size=self.num_random_feature_subset)
                self.recursive_segmenter(self.root,self.max_depth)

        def recursive_predict(self,sample,node):
                feature_index, threshold = node.split_rule
                if node.isLeaf:
                        # print("therefore the result is",node.label)
                        # print("")
                        return node.label
                else:
                        if sample[feature_index]<=threshold:
                                # print("feature_index",feature_index)
                                # print("is less than or equal to threshold",threshold)
                                return self.recursive_predict(sample,node.left)
                        else:
                                # print("feature_index",feature_index)
                                # print("is larger than threshold",threshold)
                                return self.recursive_predict(sample,node.right)
        def predict(self,data):
                pred_list = []
                for sample in data:
                        prediction = self.recursive_predict(sample,self.root)
                        pred_list.append(int(float(prediction)))
                return pred_list

class randomForest:
        def __init__(self,max_depth,num_random_feature_subset,num_trees):
```

```
                self.max_depth = max_depth
                self.num_random_feature_subset = num_random_feature_subset
                self.num_trees = num_trees
                self.tree_list = []
        def train(self,data,labels):
                for tree in range(self.num_trees):
                        data,labels = shuffle(data,labels)
                        tree = decisionTree(self.max_depth,self.num_random_feature_subset)
                        tree.train(data[data.shape[0]//5:,:],labels[data.shape[0]//5:])
                        self.tree_list.append(tree)
        def predict(self,data):
                pred_list = []
                for tree in self.tree_list:
                        a_pred = tree.predict(data)
                        pred_list.append(a_pred)
                pred_list = np.asarray(pred_list)
                final_pred = []
                for pred_index in range(pred_list.shape[1]):
                        sample_pred = mode(pred_list[:,pred_index])[0][0]
                        final_pred.append(sample_pred)
                return final_pred


def q7_a():
        def visualize_node(node,curr_depth):
                if not node.isLeaf:
                        visualize_node(node.left,curr_depth+1)
                        print("                                                "*curr_depth,"threshold:
",node.split_rule[1])
                        print("                                                  "*curr_depth,"feature:
",train_key_list[node.split_rule[0]])
                        visualize_node(node.right,curr_depth+1)
                else:
                        print("                        "*curr_depth,"this leaf's label:",node.label)
```

```
	categorical_classes = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']
	training_list = csv.DictReader(open("hw5_census_dist/train_data.csv"))
	training_dict = categorical_preprocessing(training_list,categorical_classes)
	training_labels = training_dict["label"]
	training_dict.pop("label")

	train_key_list = []
	for key in training_dict.keys():
		train_key_list.append(key)
	train_key_list.sort()

	training_set = np.asarray([training_dict[key] for key in train_key_list]).T
	#print("training et shape",training_set.shape) # 32724x113

	max_depth = 5
	#num_random_feature_subset = round(sqrt(training_set.shape[1]))
	num_random_feature_subset = training_set.shape[1]
	training_set,training_labels = shuffle(training_set,training_labels)
	t_set = training_set[:,:]
	t_labels = training_labels[:]

	titanic_classifier = decisionTree(max_depth,num_random_feature_subset)
	titanic_classifier.train(t_set,t_labels)
	visualize_node(titanic_classifier.root,0)



def spam_DT():
	spam_data = loadmat('hw5_spam_dist/dist/spam_data.mat')
	spam_training_data = spam_data['training_data']
	spam_test_data = spam_data['test_data']
```

```
	spam_training_labels = spam_data['training_labels']
	max_depth = 30
	num_random_feature_subset = spam_training_data.shape[1]
	spam_training_set = np.concatenate((spam_training_data,spam_training_labels.T),axis=1)
	np.random.shuffle(spam_training_set)
	v_set = spam_training_set[:spam_training_set.shape[0]//5,:spam_training_set.shape[1]-1]
	v_labels = spam_training_set[:spam_training_set.shape[0]//5,spam_training_set.shape[1]-1]
	t_set = spam_training_set[spam_training_set.shape[0]//5:,:spam_training_set.shape[1]-1]
	t_labels = spam_training_set[spam_training_set.shape[0]//5:,spam_training_set.shape[1]-1]
	spam_classifier = decisionTree(max_depth,num_random_feature_subset)
	spam_classifier.train(t_set,t_labels)
	v_predictions = spam_classifier.predict(v_set)
	t_predictions = spam_classifier.predict(t_set)
	#print("pred len",len(predictions))
	#print("labels len",len(v_labels))
	t_score = compute_score(t_labels,t_predictions)
	v_score = compute_score(v_labels,v_predictions)
	print("spam DT t score:",t_score)
	print("spam DT v score:",v_score)

def spam_RF():
	spam_data = loadmat('hw5_spam_dist/dist/spam_data.mat')
	spam_training_data = spam_data['training_data']
	spam_test_data = spam_data['test_data']
	spam_training_labels = spam_data['training_labels']
	max_depth = 20
	num_trees = 40
	v_pred_list = []
	num_random_feature_subset = round(sqrt(spam_training_data.shape[1]))
	spam_training_set = np.concatenate((spam_training_data,spam_training_labels.T),axis=1)
	np.random.shuffle(spam_training_set)
	v_set = spam_training_set[:spam_training_set.shape[0]//5,:spam_training_set.shape[1]-1]
```

```
        v_labels = spam_training_set[:spam_training_set.shape[0]//5,spam_training_set.shape[1]-1]
        t_set = spam_training_set[spam_training_set.shape[0]//5:,:spam_training_set.shape[1]-1]
        t_labels = spam_training_set[spam_training_set.shape[0]//5:,spam_training_set.shape[1]-1]

        spam_classifier = randomForest(max_depth,num_random_feature_subset,num_trees)
        spam_classifier.train(t_set,t_labels)
        for tree in spam_classifier.tree_list:
                print(tree.root.split_rule)
        # v_labels = [v_labels[0]]
        # v_set = [v_set[0,:]]
        v_predictions = spam_classifier.predict(v_set)
        t_predictions = spam_classifier.predict(t_set)
        v_labels = np.asarray([int(float(val)) for val in v_labels])
        t_labels = np.asarray([int(float(val)) for val in t_labels])
        v_score = compute_score(v_labels,v_predictions)
        t_score = compute_score(t_labels,t_predictions)
        print("spam RF t score:",t_score)
        print("spam RF v score:",v_score)

def spam_kaggle():
        spam_data = loadmat('hw5_spam_dist/dist/spam_data.mat')
        spam_training_data = spam_data['training_data']
        spam_test_data = spam_data['test_data']
        spam_training_labels = spam_data['training_labels']
        max_depth = 30
        num_random_feature_subset = spam_training_data.shape[1]
        spam_training_set = np.concatenate((spam_training_data,spam_training_labels.T),axis=1)
        np.random.shuffle(spam_training_set)
        t_set = spam_training_set[:,:spam_training_set.shape[1]-1]
        t_labels = spam_training_set[:,spam_training_set.shape[1]-1]
        spam_classifier = decisionTree(max_depth,num_random_feature_subset)
        spam_classifier.train(t_set,t_labels)
```

```
    test_pred = spam_classifier.predict(spam_test_data)
    spam_table = {"Category":test_pred,"Id":np.arange(0,len(test_pred))}
    spam_output = pd.DataFrame(data=spam_table)
    spam_output.to_csv("kaggle_ycls_hw5_spam.csv", index=False)
    print("spam csv created")

def toy1():
    from sklearn.utils import shuffle
    X_toy = np.asarray([[1], [2], [3], [4], [5], [6]])
    y_toy = np.asarray([0,0,0,1,1,1])
    X_toy, y_toy = shuffle(X_toy, y_toy)
    toy_tree = decisionTree(3, 1)
    toy_tree.train(X_toy, y_toy)
    print("X:",X_toy)
    print("y:",y_toy)
    pred = toy_tree.predict(X_toy)
    print("pred:",pred)
    score = compute_score(y_toy,pred)
    print("score:",score)

def toy2():
    example_training_set = np.asarray([[14,70,100,1000,10000],
                                       [2,20,200,8000,200],
                                       [3,30,300,3000,30000],
                                       [4,40,400,4000,40000],
                                       [5,50,500,5000,50000]])
    example_training_labels = np.asarray([0,0,0,1,1])
    example_t_set, example_t_labels = shuffle(example_training_set,example_training_labels)
    example_v_set = np.asarray([[j*(10**i) for i in range(5)]for j in range(2,7)])
    print("example v set:")
    print(example_v_set)
    print("example t set",example_t_set)
```

```
        print("example t labels",example_t_labels)
        example_v_labels = np.asarray([0,0,1,1,1])
        max_depth = 3
        num_random_feature_subset = example_t_set.shape[1]
        example_classifier = decisionTree(max_depth,num_random_feature_subset)
        example_classifier.train(example_t_set,example_t_labels)
        example_pred = example_classifier.predict(example_v_set)
        print("example pred",example_pred)
        score = compute_score(example_v_labels,example_pred)
        print("score:",score)

def categorical_preprocessing(data,categorical_classes,dropping_classes=[]):
        category_dict = {}
        for category in data.fieldnames:
                category_dict[category] = []
        #num_samples = len(data)
        for sample in data:
                for category in data.fieldnames:
                        category_dict[category].append(sample[category])
        for drop_class in dropping_classes:
                category_dict.pop(drop_class)
        #print("categ dict",category_dict)
        new_features = {}
        for feature, feature_vals in category_dict.items():
                if feature in categorical_classes:
                        #print("feature vals",feature_vals)
                        unique_feature_vals = np.unique(feature_vals)
                        mode_feature = mode(feature_vals)[0][0]
                        unique_feature_vals = unique_feature_vals.tolist()
                        if '?' in unique_feature_vals:
                                unique_feature_vals.remove('?')
                        if '' in unique_feature_vals:
```

```
                unique_feature_vals.remove('')
            unique_feature_vals = np.asarray(unique_feature_vals)
            #print("unique feature vals for feature",feature)
            #print(unique_feature_vals)
            for new_feature in unique_feature_vals:
                new_features[new_feature] = [0 for _ in feature_vals]
            for new_feature in unique_feature_vals:
                for idx in range(len(new_features[new_feature])):
                    if feature_vals[idx]==new_feature:
                        new_features[new_feature][idx] = 1
                    elif feature_vals[idx]=='?' or feature_vals[idx]=='':
                        new_features[mode_feature][idx] = 1
    for category in categorical_classes:
        if category not in dropping_classes:
            category_dict.pop(category)
    category_dict.update(new_features)
    return category_dict


def census(max_depth=10):
    census_categorical_classes = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race',
'sex', 'native-country']
    census_list = csv.DictReader(open("hw5_census_dist/train_data.csv"))
    census_dict = categorical_preprocessing(census_list,census_categorical_classes)
    training_labels = census_dict["label"]
    census_dict.pop("label")

    train_key_list = []
    for key in census_dict.keys():
        train_key_list.append(key)
    train_key_list.sort()
    #print("sorted train key",train_key_list)

    training_set = np.asarray([census_dict[key] for key in train_key_list]).T
```

```
        #print("training et shape",training_set.shape) # 32724x113
        #print("test set shape",test_set.shape) # 16118x113

        #max_depth = 10
        #num_random_feature_subset = round(sqrt(training_set.shape[1]))
        num_random_feature_subset = training_set.shape[1]
        training_set,training_labels = shuffle(training_set,training_labels)
        t_set = training_set[training_set.shape[0]//5:,:]
        t_labels = training_labels[training_set.shape[0]//5:]
        v_set = training_set[:training_set.shape[0]//5,:]
        v_labels = training_labels[:training_set.shape[0]//5]

        census_classifier = decisionTree(max_depth,num_random_feature_subset)
        census_classifier.train(t_set,t_labels)
        v_predictions = census_classifier.predict(v_set)
        v_labels = np.asarray([int(float(val)) for val in v_labels])
        v_score = compute_score(v_labels,v_predictions)
        t_predictions = census_classifier.predict(t_set)
        t_labels = np.asarray([int(float(val)) for val in t_labels])
        t_score = compute_score(t_labels,t_predictions)
        print("census DT t_score:",t_score)
        print("census DT v_score:",v_score)
        return v_score


def census_RF(max_depth=10):
        census_categorical_classes = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']
        census_list = csv.DictReader(open("hw5_census_dist/train_data.csv"))
        census_dict = categorical_preprocessing(census_list,census_categorical_classes)
        training_labels = census_dict["label"]
        census_dict.pop("label")

        train_key_list = []
```

```
for key in census_dict.keys():
        train_key_list.append(key)
train_key_list.sort()

training_set = np.asarray([census_dict[key] for key in train_key_list]).T
#print("training et shape",training_set.shape) # 32724x113
#print("test set shape",test_set.shape) # 16118x113

#max_depth = 10
num_random_feature_subset = round(sqrt(training_set.shape[1]))
#num_random_feature_subset = training_set.shape[1]
num_trees = 10
training_set,training_labels = shuffle(training_set,training_labels)
t_set = training_set[training_set.shape[0]//5:,:]
t_labels = training_labels[training_set.shape[0]//5:]
v_set = training_set[:training_set.shape[0]//5,:]
v_labels = training_labels[:training_set.shape[0]//5]

census_classifier = randomForest(max_depth,num_random_feature_subset,num_trees)
census_classifier.train(t_set,t_labels)

# v_labels = np.asarray([v_labels[0]])
# v_set = np.asarray([v_set[0,:]])

v_predictions = census_classifier.predict(v_set)
v_labels = np.asarray([int(float(val)) for val in v_labels])
v_score = compute_score(v_labels,v_predictions)
t_predictions = census_classifier.predict(t_set)
t_labels = np.asarray([int(float(val)) for val in t_labels])
t_score = compute_score(t_labels,t_predictions)
print("census RF t_score:",t_score)
print("census RF v_score:",v_score)
```

```
	return v_score


def census_kaggle():
	census_categorical_classes = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']
	census_list = csv.DictReader(open("hw5_census_dist/train_data.csv"))
	test_census_list = csv.DictReader(open("hw5_census_dist/test_data.csv"))
	#num_test_samples = len([0 for _ in test_census_list])
	census_dict = categorical_preprocessing(census_list,census_categorical_classes)
	test_census_dict = categorical_preprocessing(test_census_list,census_categorical_classes)
	training_labels = census_dict["label"]
	census_dict.pop("label")

	train_key_list = []
	for key in census_dict.keys():
		train_key_list.append(key)
	train_key_list.sort()

	test_key_list = []
	for key in test_census_dict.keys():
		test_key_list.append(key)
	test_key_list.sort()

	training_set = np.asarray([census_dict[key] for key in train_key_list]).T
	test_set = []
	for key in train_key_list:
		if key in test_key_list:
			test_set.append(test_census_dict[key])
		else:
			test_set.append([0 for _ in range(16118)])
	test_set = np.asarray(test_set).T

	max_depth = 10
```

```
	#num_random_feature_subset = round(sqrt(training_set.shape[1]))
	num_random_feature_subset = training_set.shape[1]
	training_set,training_labels = shuffle(training_set,training_labels)
	t_set = training_set
	t_labels = training_labels

	census_classifier = decisionTree(max_depth,num_random_feature_subset)
	census_classifier.train(t_set,t_labels)
	test_pred = census_classifier.predict(test_set)
	census_table = {"Category":test_pred,"Id":np.arange(1,len(test_pred)+1)}
	census_output = pd.DataFrame(data=census_table)
	census_output.to_csv("kaggle_ycls_hw5_census.csv", index=False)
	print("census csv file created")

def q6_d():
	max_depth_list = range(1,5)
	score_list = []
	for max_depth in max_depth_list:
		score_list.append(census(max_depth))
	#print("score_list:",score_list)
	plt.plot(max_depth_list,score_list,'b-')
	plt.title('Max Depth vs. Accuracy')
	plt.xlabel('Max Depth')
	plt.ylabel('Accuracy Rate')
	plt.show()

def titanic():
	training_list = csv.DictReader(open("hw5_titanic_dist/titanic_training.csv"))
	categorical_classes = ['sex','ticket','cabin','embarked']
	dropping_classes = ['ticket','cabin']
	training_dict = categorical_preprocessing(training_list,categorical_classes,dropping_classes)
	training_labels = training_dict["survived"]
```

```
        training_dict.pop("survived")

        train_key_list = []
        for key in training_dict.keys():
                train_key_list.append(key)
        train_key_list.sort()

        training_set = np.asarray([training_dict[key] for key in train_key_list]).T
        #print("training et shape",training_set.shape) # 32724x113
        #print("test set shape",test_set.shape) # 16118x113

        max_depth = 10
        #num_random_feature_subset = round(sqrt(training_set.shape[1]))
        num_random_feature_subset = training_set.shape[1]
        training_set,training_labels = shuffle(training_set,training_labels)
        t_set = training_set[training_set.shape[0]//5:,:]
        t_labels = training_labels[training_set.shape[0]//5:]
        v_set = training_set[:training_set.shape[0]//5,:]
        v_labels = training_labels[:training_set.shape[0]//5]

        titanic_classifier = decisionTree(max_depth,num_random_feature_subset)
        titanic_classifier.train(t_set,t_labels)
        v_predictions = titanic_classifier.predict(v_set)
        v_labels = np.asarray([int(float(val)) for val in v_labels])
        v_score = compute_score(v_labels,v_predictions)
        t_predictions = titanic_classifier.predict(t_set)
        t_labels = np.asarray([int(float(val)) for val in t_labels])
        t_score = compute_score(t_labels,t_predictions)
        print("titanic DT t_score:",t_score)
        print("titanic DT v_score:",v_score)

def titanic_RF():
```

```
training_list = csv.DictReader(open("hw5_titanic_dist/titanic_training.csv"))
categorical_classes = ['sex','ticket','cabin','embarked']
dropping_classes = ['ticket','cabin']
training_dict = categorical_preprocessing(training_list,categorical_classes,dropping_classes)
training_labels = np.asarray(training_dict["survived"])
training_dict.pop("survived")

train_key_list = []
for key in training_dict.keys():
        train_key_list.append(key)
train_key_list.sort()

training_set = np.asarray([training_dict[key] for key in train_key_list]).T

max_depth = 10
num_trees = 50
num_random_feature_subset = round(sqrt(training_set.shape[1]))
#num_random_feature_subset = training_set.shape[1]
training_set,training_labels = shuffle(training_set,training_labels)
t_set = training_set[training_set.shape[0]//5:,:]
t_labels = training_labels[training_set.shape[0]//5:]
v_set = training_set[:training_set.shape[0]//5,:]
v_labels = training_labels[:training_set.shape[0]//5]
v_pred_list = []
for tree_idx in range(num_trees):
        titanic_classifier = decisionTree(max_depth,num_random_feature_subset)
        titanic_classifier.train(t_set,t_labels)
        predictions = titanic_classifier.predict(v_set)
        v_pred_list.append(predictions)
        t_set, t_labels = shuffle(t_set, t_labels)
v_pred_list = np.asarray(v_pred_list)
v_final_pred = []
```

```
        for pred_index in range(v_pred_list.shape[1]):
                final_pred = mode(v_pred_list[:,pred_index])[0][0]
                v_final_pred.append(final_pred)
        v_labels = np.asarray([int(float(val)) for val in v_labels])
        v_score = compute_score(v_labels,v_final_pred)
        print("score:",v_score)

def titanic_implemented_RF():
        training_list = csv.DictReader(open("hw5_titanic_dist/titanic_training.csv"))
        categorical_classes = ['sex','ticket','cabin','embarked']
        dropping_classes = ['ticket','cabin']
        training_dict = categorical_preprocessing(training_list,categorical_classes,dropping_classes)
        training_labels = training_dict["survived"]
        training_dict.pop("survived")

        train_key_list = []
        for key in training_dict.keys():
                train_key_list.append(key)
        train_key_list.sort()



        training_set = np.asarray([training_dict[key] for key in train_key_list]).T

        max_depth = 10
        num_random_feature_subset = round(sqrt(training_set.shape[1]))
        #num_random_feature_subset = training_set.shape[1]
        num_trees = 10
        training_set,training_labels = shuffle(training_set,training_labels)
        t_set = training_set[training_set.shape[0]//5:,:]
        t_labels = training_labels[training_set.shape[0]//5:]
        v_set = training_set[:training_set.shape[0]//5,:]
        v_labels = training_labels[:training_set.shape[0]//5]
```

```
	titanic_classifier = randomForest(max_depth,num_random_feature_subset,num_trees)
	titanic_classifier.train(t_set,t_labels)
	v_predictions = titanic_classifier.predict(v_set)
	v_labels = np.asarray([int(float(val)) for val in v_labels])
	v_score = compute_score(v_labels,v_predictions)
	t_predictions = titanic_classifier.predict(t_set)
	t_labels = np.asarray([int(float(val)) for val in t_labels])
	t_score = compute_score(t_labels,t_predictions)
	print("titanic RF t_score:",t_score)
	print("titanic RF v_score:",v_score)

def titanic_kaggle():
	training_list = csv.DictReader(open("hw5_titanic_dist/titanic_training.csv"))
	test_list = csv.DictReader(open("hw5_titanic_dist/titanic_testing_data.csv"))
	categorical_classes = ['sex','ticket','cabin','embarked']
	dropping_classes = ['ticket','cabin']
	training_dict = categorical_preprocessing(training_list,categorical_classes,dropping_classes)
	test_dict = categorical_preprocessing(test_list,categorical_classes,dropping_classes)
	training_labels = training_dict["survived"]
	training_dict.pop("survived")

	train_key_list = []
	for key in training_dict.keys():
		train_key_list.append(key)
	train_key_list.sort()

	test_key_list = []
	for key in test_dict.keys():
		test_key_list.append(key)
	test_key_list.sort()
```

```
training_set = np.asarray([training_dict[key] for key in train_key_list]).T
test_set = []
for key in train_key_list:
        if key in test_key_list:
                test_set.append(test_dict[key])
        else:
                test_set.append([0 for _ in range(16118)])
test_set = np.asarray(test_set).T
print("training set shape",training_set.shape)
print("test set shape",test_set.shape)


max_depth = 20
num_trees = 50
num_random_feature_subset = round(sqrt(training_set.shape[1]))
#num_random_feature_subset = training_set.shape[1]
training_set,training_labels = shuffle(training_set,training_labels)
t_set = training_set[:,:]
t_labels = training_labels[:]
test_pred_list = []
for tree_idx in range(num_trees):
        titanic_classifier = decisionTree(max_depth,num_random_feature_subset)
        titanic_classifier.train(t_set,t_labels)
        predictions = titanic_classifier.predict(test_set)
        test_pred_list.append(predictions)
        t_set, t_labels = shuffle(t_set, t_labels)
test_pred_list = np.asarray(test_pred_list)
test_final_pred = []
for pred_index in range(test_pred_list.shape[1]):
        final_pred = mode(test_pred_list[:,pred_index])[0][0]
        test_final_pred.append(int(final_pred))
titanic_table = {"Category":test_final_pred,"Id":np.arange(1,len(test_final_pred)+1)}
titanic_output = pd.DataFrame(data=titanic_table)
```

```
        titanic_output.to_csv("kaggle_ycls_hw5_titanic.csv", index=False)
        print("titanic csv file created")

main()
```