CS 289: Introduction to Machine Learning
Homework 5: Decision Trees & Random Forests

Shuhui Huang SID:3032129712

1. **Implement decision trees**

To build the decision tree, first we have to define node, including leaf node, left node, right node and so on. After that, we can go on build decision tree.

After initiating the tree, we first define entropy of an index set S is the average surprise: $\mathrm{H}(\mathrm{s}) = -\sum_c p_c log_2 p_c$.

Since we have defined the entropy function, we then compute weighted average entropy after split, which is $\mathrm{H}_{after} = \frac{|S_l|H(S_l)+|S_r|H(S_r)}{|S_l|+|S_r|}$.

Now we have to define the splitting value, or say, thresholds. We use the mean of feature values mean. If the mean of each feature is $\overline{X_0}, \overline{X_1}, \ldots, \overline{X_n}$ the threshold will be $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} \overline{X_\iota}$. And we iteratively choosing feature j and splitting value, computing their weighted average entropy after split, and choose the best splitting feature j and splitting value β that minimizes weighted average $\mathrm{H}_{after} = \frac{|S_l|H(S_l)+|S_r|H(S_r)}{|S_l|+|S_r|}$.

After we have the best splitting feature j and splitting value β, we can grow the tree. The way is:

GrowTree(S)

if (y_i = C for all i ∈ S and some class C) then{

return new leaf(C) [We say the leaves are "pure"]

} else {

choose best splitting feature j and splitting value β (*)

$S_l = \{i: X_{ij} < \beta\}$

$S_r = \{i: X_{ij} \geq \beta\}$

return new node (j, β, GrowTree(S_l), GrowTree(S_r))

}

Now, we have the function to grow tree and split the data. Our next step is to traverse the tree while using split rule at each node. And using test data, we can make prediction accordingly.

My code is:

```
class Node():
    def __init__(self,my_rule,my_left,my_right):
        self.split_rule = my_rule
        self.left = my_left
        self.right = my_right
    def is_leaf(self):
        return False
    def rule(self):
        return self.split_rule
    def left_node(self):
        return self.left
    def right_node(self):
        return self.right

class Leaf_Node():
    def __init__(self,my_label):
        self.label = my_label
    def is_leaf(self):
        return True
    def mylabel(self):
        return self.label

## build decision tree##
class DecisionTree():
    def __init__(self, my_max_depth, my_num_feature):
        self.max_depth = my_max_depth
        self.num_feature = my_num_feature
        self.root = Node(None,None,None)
    def entropy(self, prob):
        if prob<0 or prob>1:
            print("Wrong probability!")
            return None
        elif prob==0:
            return 0
        elif prob==1:
            return 0
        else:
```

```
        return -prob*log(prob,2)-(1-prob)*log(1-prob,2)


    def impurity(self, left_labels, right_labels):
        ### compute the weighted average entropy of the children ###
        left_count = float(len(left_labels))
        right_count = float(len(right_labels))
        if left_count>0 and right_count>0:
            left_prob = sum([x==0 for x in left_labels])/left_count
            left_entropy = self.entropy(left_prob)
            right_prob = sum([x==0 for x in right_labels])/right_count
            right_entropy = self.entropy(right_prob)
            return
(left_count/(left_count+right_count))*left_entropy+(right_count/(left_count+right_count))*rig
ht_entropy
        elif left_count==0:
            right_prob = sum([x==0 for x in right_labels])/right_count
            right_entropy = self.entropy(right_prob)
            return right_entropy
        elif right_count==0:
            left_prob = sum([x==0 for x in left_labels])/left_count
            left_entropy = self.entropy(left_prob)
            return left_entropy

    def segmenter(self,data,labels,num_features):
        ### thresholds are average of feature's means ###
        data1 = data[labels==1]
        data0 = data[labels==0]
        mean1 = np.mean(data1,axis=0)
        mean0 = np.mean(data0,axis=0)
        threshold_list = (mean0+mean1)/2
        feature_len = len(data[0])
        feature_collection = random.sample(range(feature_len),num_features)
        impurity_list = []
        for i in feature_collection:
            threshold = threshold_list[i]
            group1_labels = labels[data[:,i]>=threshold]
            group2_labels = labels[data[:,i]< threshold]
            impurity_list.append(self.impurity(group2_labels,group1_labels))
        best_feature_index = feature_collection[impurity_list.index(min(impurity_list))]
        best_feature_threshold = threshold_list[best_feature_index]
        return (best_feature_index,best_feature_threshold)

    def GrowTree(self,data,labels,depth=1):
```

```
    ### recursively grow a tree by constructing nodes ###
    if depth<=self.max_depth:
        if len(np.unique(labels))==1:
            ### all labels are the same ###
            return Leaf_Node(np.unique(labels)[0])
        else:
            segment = self.segmenter(data,labels,self.num_feature)
            index = segment[0]
            threshold = segment[1]
            left = data[:,index]<threshold
            right = data[:,index]>=threshold
            left_data = data[left]
            right_data = data[right]
            left_labels = labels[left]
            right_labels =labels[right]
            if len(left_labels)==0 or len(right_labels)==0:
                ### fail to find a split to reduce impurity ###
                common_label = Counter(labels).most_common(1)[0][0]
                return Leaf_Node(common_label)
            else:
                left_node = self.GrowTree(left_data,left_labels,depth+1)
                right_node = self.GrowTree(right_data,right_labels,depth+1)
                return Node((index,threshold),left_node,right_node)
    else:
        label = Counter(labels).most_common(1)[0][0]
        return Leaf_Node(label)

def train(self,data,labels):
    self.root = self.GrowTree(data,labels)

def TraverseTree(self,root,X):
    ##recursively traverse the tree##
    if root.is_leaf()==True:
        return root.mylabel()
    else:
        index = root.rule()[0]
        threshold = root.rule()[1]
        if X[index]<threshold:
            return self.TraverseTree(root.left_node(),X)
        else:
            return self.TraverseTree(root.right_node(),X)

def predict(self,test_data):
    predicted_result = []
```

```
for X in test_data:
    predicted_result.append(self.TraverseTree(self.root,X))
return predicted_result
```

2. **Implement random forests.**
   Because we have already written the function of decision tree. So we only need to initiate the random forest, then write the predict and train function.

   The basic idea is: at each split, take random sample of m features (out of d). Choose best split from m features. Different random sample for each split.

   My code is:

```
class RandomForest():
    # num_tree: how many trees to grow
    # num_sample: how many random samples used to train each tree
    # num_feature: how many features to select from for each node
    # max_depth: maximum depth for each tree

    def __init__(self,num_tree,num_sample,num_feature,max_depth):
        self.num_tree = num_tree
        self.num_sample = num_sample
        self.num_feature = num_feature
        self.max_depth = max_depth
        self.trees=[]

    def train(self,data,labels):
        for i in range(self.num_tree):
            sample_index = np.random.choice(range(len(data)),self.num_sample,replace=True)
            train_data = data[sample_index]
            train_label = labels[sample_index]
            sub_tree = DecisionTree(self.max_depth,self.num_feature)
            sub_tree.train(train_data,train_label)
            self.trees.append(sub_tree)

    def predict(self,test_data):
        prediction_list = []
        for t in self.trees:
            prediction_list.append(t.predict(test_data))
        return (np.mean(np.array(prediction_list),axis=0)>0.5).astype(int)
```

3. Describe implementation details.

(a) How did you deal with categorical features and missing values?

As can be seen in my code, for the missing features in the data, I replaced the “?” with the mode of the feature in the training data as an approximation. And for the categorical variables, I make the use of the “one-hot encoding” as suggested in the appendix.

(b) What was your stopping criteria?

My stopping criteria is: setting the average entropy of the children nodes as the splitting criteria, and stop growing the tree if it reaches the maximum depth or we cannot find a split to reduce the impurity for the node.

(c) Did you do anything special to speed up training?

Because I use cross validation to decide the hyper-parameters, which including num_tree, num_sample, num_feature, max_depth. It would take too long to train them together, so I trained them separately as an approximation, and it can save time.

(d) How did you implement random forests?

For random forests, I modified the decision tree class by adding number of feature as a variable so that it can be used directly in the random forest class. Then I use cross-validation to get the best set of parameters and then trained my data set.

(e) Anything else cool you implemented?

One important thing I noticed is that for "Native Country" in census data set, there is "Holand-Netherlands" in the training dataset, but it does not exist in the testing dataset. So, it will cause the feature length of the vectored training data be different from that of the testing data. To solve this problem, I just changed it into "United States" for simplicity because there only one record for “Holand-Neitherland”.

4. Performance evaluation.

	Spam (training accuracy)	Spam (validation accuracy)	Census (training accuracy)	Census (validation accuracy)	Titanic (training accuracy)	Titanic (validation accuracy)
Decision Tree	0.9576	0.8711	0.8705	0.8461	0.9287	0.7950
Random Forest	0.8773	0.8684	0.8554	0.8480	0.9425	0.8263
Kaggle score	0.86320		0.84762		0.81935	

5. Write up requirements for the spam dataset

(a) Feature Selection

As what I did in Homework 1, I added some features into the dataset such as the frequency of the words: “medication”, “visit”, “link”, “dislike”, “pay less”, “save”, “free”, “discount”, “unsubscribe”, “offer”, “free”, “deal”, “sex”, which tend to appear more in the spam emails.

(b) For your decision tree, state the splits.
For my first observation of the training data set, the split is:

(exclamation)<1.0
(http) >=1.0
(energy)<2.0
(para)>=2.0
(million)<1.0
(energy)<1.0
(remove)<1.0
(transfer)<1.0
(med)<1.0
(discreet)<1.0
(discover)<1.0
(weight)<1.0
(differ)<1.0
(other)<1.0
(height)<1.0
(off)<1.0
(confirm)<1.0
(?)<35.0
(premium)<1.0
(money back) <1.0

Therefore, this email was ham.

(c)For random forests, find and state the most common splits made at the root node of the trees.

for a 200-tree random forest (the parameters are: num_tree: 200, num_sample: 800, num_feature: 8, max_depth: 20), the most common splits made at the root nodes are:

(exclamation)<1 (17)
(http)<1 (17)
(sex)<1 (15)
(med)<1 (14)
(money)<1 (11)
(save)<1 (8)
(viagra)<1 (8)
(energy)<1 (7)
(featured)<1 (6)
(meter)<1 (6)

6. Write up requirements for the census dataset

(a)

To preprocess the data, for the missing features in the data, I replaced the “?” with the mode of the feature in the training data as an approximation. And for the categorical variables, I make use of the “one-hot encoding” as suggested in the appendix.

Moreover, for "Native Country" in the data set, there is "Holand-Netherlands" in the training dataset, but it does not exist in the testing dataset. So, it will cause the feature length of the vectored training data be different from that of the testing data. To solve this problem, I just changed it into "United States" for simplicity because there only one record for “Holand-Neitherland”.

(b) For your decision tree, state the splits.
For my first observation of the training data set, the split is:

(marital-status=Married-civ-spouse) >=1.0
(education-num) >10.0
(capital-gain) <2453.0
(education-num) < 14.0
(education-num) > 12.0
(capital-loss)< 164.0
(hours-per-week)>43.0
(workclass=Self-emp-not-inc) <1.0
(occupation=Exec-managerial) >=1.0
(age) >=41.0

Therefore, this people’s label is 0.

(c) for a 100-tree random forest (the parameters are: num_tree: 100, num_sample: 800, num_feature:30, max_depth: 10), the most common splits made at the root nodes are:

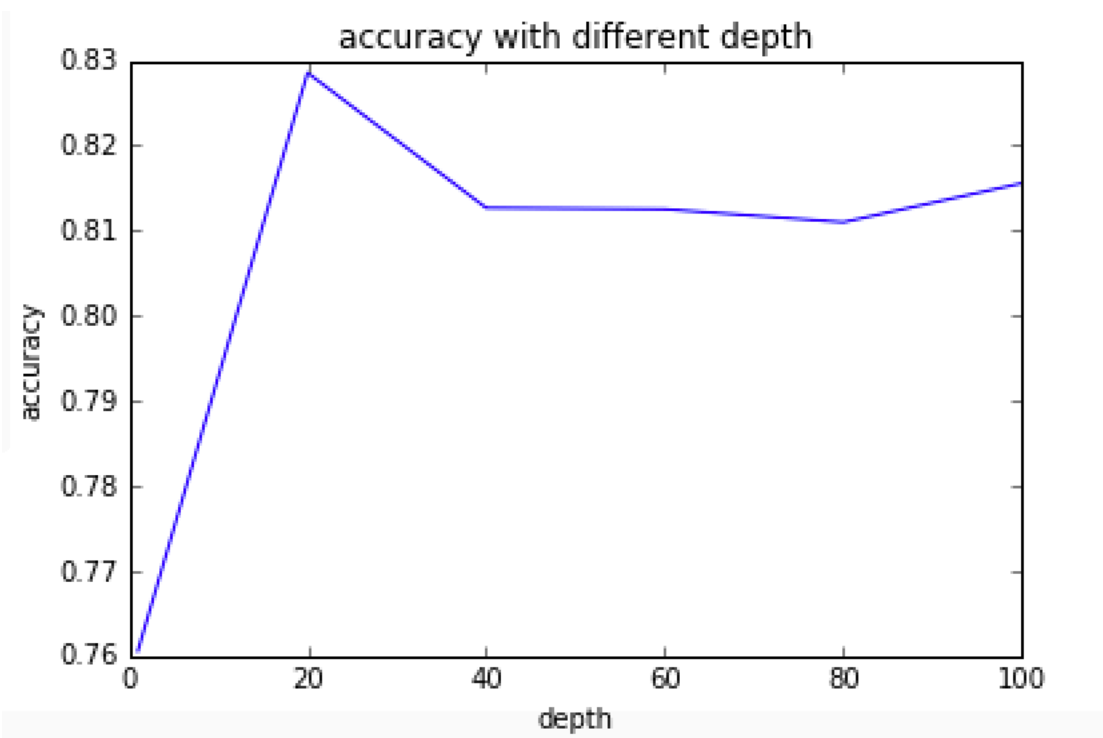
(relationship=Own-child) < 0.108950779471 (2)
(marital-status=Married-civ-spouse) < 0.583119461125 (1)
(marital-status=Married-civ-spouse) < 0.573129739701 (1)

(marital-status=Married-civ-spouse) < 0.598915075415 (1)
(marital-status=Never-married) < 0.230456761148 (1)
(marital-status=Never-married) < 0.221106166561 (1)
(marital-status=Never-married) < 0.23379477038 (1)
(capital-gain) < 2024.06825911 (1)
(marital-status=Married-civ-spouse) < 0.551809210526 (1)
(relationship=Husband) < 0.549068389139 (1)

(d) I have generated a random 80/20 training/validation split. Train decision trees with varying maximum depths going from depth = 1 to depth = 100 with all other hyper parameters fixed. And the plot is:

I find at first validation accuracy will increase with the depth increase, when depth reach a certain value, the accuracy will no longer increase with the depth increase. Because when max_depth reach a certain value, continue increasing depth may lead to overfitting.

The best depth is 20, which gives the best accuracy.



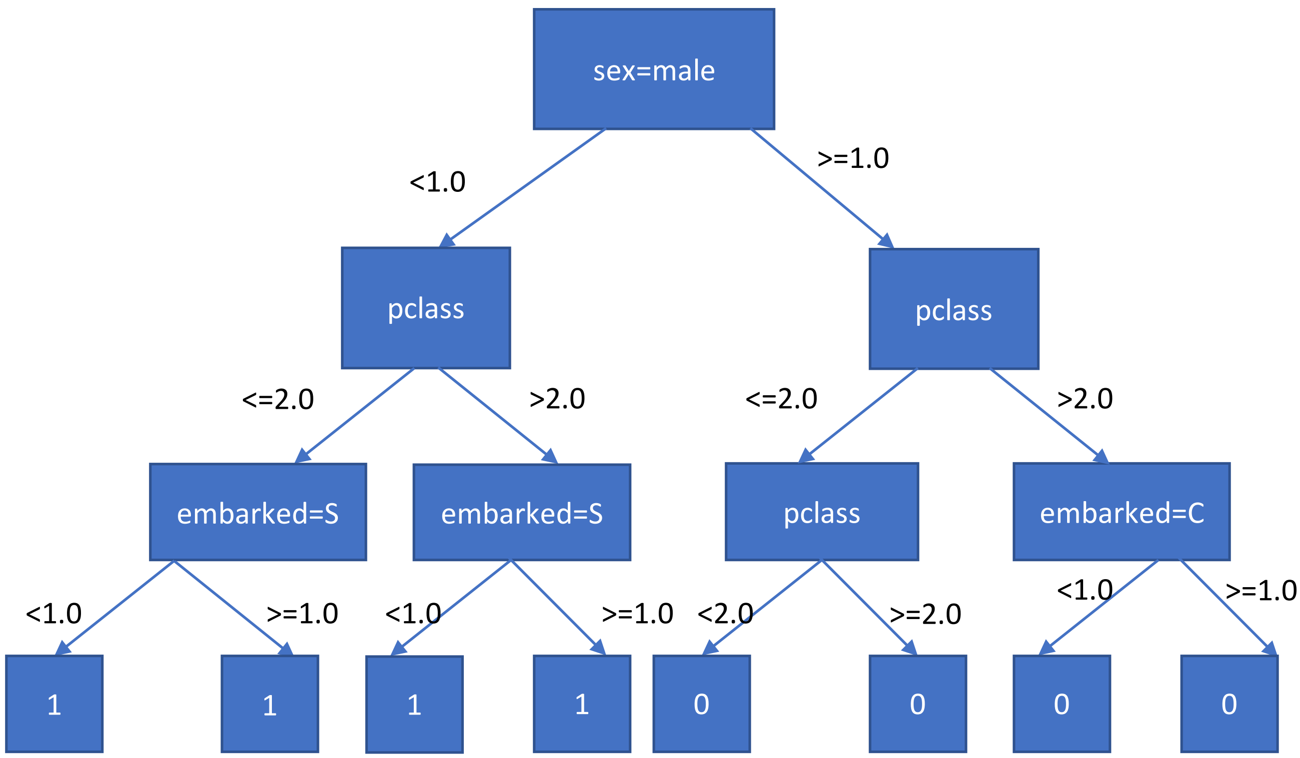
7. **Write up requirements for the Titanic dataset**

(a)

For the missing value in Titanic dataset, I replaced it with mode.

Another important thing is: I delete the column 'ticket' and 'cabin' because each one's data is different, and it will cause training data and testing data have a huge difference in dimension.

And my decision tree has been plotted as the below:



Appendix: code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 19 22:26:22 2017

@author: huangshuhui
"""

###problem1: decision tree###
import scipy.io
import random
import numpy as np
from math import log as log
from collections import Counter
import csv
import matplotlib.pyplot as plt
from sklearn.feature_extraction import DictVectorizer
import pandas as pd
from sklearn.preprocessing import Imputer

## define nodes##

class Node():
    def __init__(self,my_rule,my_left,my_right):
        self.split_rule = my_rule
        self.left = my_left
        self.right = my_right
    def is_leaf(self):
        return False
    def rule(self):
        return self.split_rule
    def left_node(self):
        return self.left
    def right_node(self):
```

```
        return self.right

class Leaf_Node():
    def __init__(self,my_label):
        self.label = my_label
    def is_leaf(self):
        return True
    def mylabel(self):
        return self.label

## build decision tree##
class DecisionTree():
    def __init__(self, my_max_depth, my_num_feature):
        self.max_depth = my_max_depth
        self.num_feature = my_num_feature
        self.root = Node(None,None,None)
    def entropy(self, prob):
        if prob<0 or prob>1:
            print("Wrong probability!")
            return None
        elif prob==0:
            return 0
        elif prob==1:
            return 0
        else:
            return -prob*log(prob,2)-(1-prob)*log(1-prob,2)


    def impurity(self, left_labels, right_labels):
        ### compute the weighted average entropy of the children ###
        left_count = float(len(left_labels))
        right_count = float(len(right_labels))
        if left_count>0 and right_count>0:
            left_prob = sum([x==0 for x in left_labels])/left_count
            left_entropy = self.entropy(left_prob)
            right_prob = sum([x==0 for x in right_labels])/right_count
            right_entropy = self.entropy(right_prob)
```

```
        return
(left_count/(left_count+right_count))*left_entropy+(right_count/(left_count+righ
t_count))*right_entropy
      elif left_count==0:
        right_prob = sum([x==0 for x in right_labels])/right_count
        right_entropy = self.entropy(right_prob)
        return right_entropy
      elif right_count==0:
        left_prob = sum([x==0 for x in left_labels])/left_count
        left_entropy = self.entropy(left_prob)
        return left_entropy

    def segmenter(self,data,labels,num_features):
      ### thresholds are average of feature's means ###
      data1 = data[labels==1]
      data0 = data[labels==0]
      mean1 = np.mean(data1,axis=0)
      mean0 = np.mean(data0,axis=0)
      threshold_list = (mean0+mean1)/2
      feature_len = len(data[0])
      feature_collection = random.sample(range(feature_len),num_features)
      impurity_list = []
      for i in feature_collection:
        threshold = threshold_list[i]
        group1_labels = labels[data[:,i]>=threshold]
        group2_labels = labels[data[:,i]< threshold]
        impurity_list.append(self.impurity(group2_labels,group1_labels))
      best_feature_index =
feature_collection[impurity_list.index(min(impurity_list))]
      best_feature_threshold = threshold_list[best_feature_index]
      return (best_feature_index,best_feature_threshold)

    def GrowTree(self,data,labels,depth=1):
      ### recursively grow a tree by constructing nodes ###
      if depth<=self.max_depth:
        if len(np.unique(labels))==1:
          ### all labels are the same ###
```

```
            return Leaf_Node(np.unique(labels)[0])
        else:
            segment = self.segmenter(data,labels,self.num_feature)
            index = segment[0]
            threshold = segment[1]
            left = data[:,index]<threshold
            right = data[:,index]>=threshold
            left_data = data[left]
            right_data = data[right]
            left_labels = labels[left]
            right_labels =labels[right]
            if len(left_labels)==0 or len(right_labels)==0:
                ### fail to find a split to reduce impurity ###
                common_label = Counter(labels).most_common(1)[0][0]
                return Leaf_Node(common_label)
            else:
                left_node = self.GrowTree(left_data,left_labels,depth+1)
                right_node = self.GrowTree(right_data,right_labels,depth+1)
                return Node((index,threshold),left_node,right_node)
    else:
        label = Counter(labels).most_common(1)[0][0]
        return Leaf_Node(label)

def train(self,data,labels):
    self.root = self.GrowTree(data,labels)

def TraverseTree(self,root,X):
    ##recursively traverse the tree##
    if root.is_leaf()==True:
         return root.mylabel()
    else:
        index = root.rule()[0]
        threshold = root.rule()[1]
        if X[index]<threshold:
             return self.TraverseTree(root.left_node(),X)
        else:
            return self.TraverseTree(root.right_node(),X)
```

```
    def predict(self,test_data):
        predicted_result = []
        for X in test_data:
            predicted_result.append(self.TraverseTree(self.root,X))
        return predicted_result

###problem 2: random forest###
class RandomForest():
    # num_tree: how many trees to grow
    # num_sample: how many random samples used to train each tree
    # num_feature: how many features to select from for each node
    # max_depth: maximum depth for each tree

    def __init__(self,num_tree,num_sample,num_feature,max_depth):
        self.num_tree = num_tree
        self.num_sample = num_sample
        self.num_feature = num_feature
        self.max_depth = max_depth
        self.trees=[]

    def train(self,data,labels):
        for i in range(self.num_tree):
            sample_index = 
np.random.choice(range(len(data)),self.num_sample,replace=True)
            train_data = data[sample_index]
            train_label = labels[sample_index]
            sub_tree = DecisionTree(self.max_depth,self.num_feature)
            sub_tree.train(train_data,train_label)
            self.trees.append(sub_tree)

    def predict(self,test_data):
        prediction_list = []
        for t in self.trees:
            prediction_list.append(t.predict(test_data))
        return (np.mean(np.array(prediction_list),axis=0)>0.5).astype(int)
```

###problem5: spam data###

```
train_data = scipy.io.loadmat('/Users/huangshuhui/Google
Drive/study/cs289/hw2017/hw5/data/hw5_spam_dist/spam_data_sh.mat')
test_x=train_data['test_data'].astype(float)
train_y=train_data['training_labels'][0].astype(int)
train_x=train_data['training_data'].astype(float)
x_y = list(zip(train_x, train_y))
random.shuffle(x_y)
train_x = np.array([e[0] for e in x_y])
train_y = np.ravel([e[1] for e in x_y])

## using the cross_validation to get the best parameter##
def k_fold_cross_validation_index(length, K):
    for k in range(K):
        training_index = [x for x in range(length) if x % K != k]
        validation_index = [x for x in range(length) if x % K == k]
        yield training_index, validation_index
def decision_tree_CV_accuracy(depth):
    accuracies=[]
    for train_index,validation_index in
k_fold_cross_validation_index(len(train_x),5):
        cv_train_x=np.array([train_x[i] for i in train_index])
        cv_train_y=np.array([train_y[i] for i in train_index])
        cv_test_x=np.array([train_x[i] for i in validation_index])
        cv_test_y=np.array([train_y[i] for i in validation_index])
        tree = DecisionTree(depth,len(train_x[0]))
        tree.train(cv_train_x,cv_train_y)
        cv_predict_y=tree.predict(cv_test_x)
        accuracies.append(sum(cv_predict_y==cv_test_y)/len(cv_test_y))
    return sum(accuracies)/len(accuracies)

def
random_forest_CV_accuracy(num_tree,num_sample,num_feature,max_depth):
    accuracies=[]
```

```
    for train_index,validation_index in
k_fold_cross_validation_index(len(train_x),5):
        cv_train_x=np.array([train_x[i] for i in train_index])
        cv_train_y=np.array([train_y[i] for i in train_index])
        cv_test_x=np.array([train_x[i] for i in validation_index])
        cv_test_y=np.array([train_y[i] for i in validation_index])
        forest = RandomForest(num_tree,num_sample,num_feature,max_depth)
        forest.train(cv_train_x,cv_train_y)
        cv_predict_y=forest.predict(cv_test_x)
        accuracies.append(sum(cv_predict_y==cv_test_y)/len(cv_test_y))
    return sum(accuracies)/len(accuracies)

##test spam data set##
# find splits in decision tree (spam) #
ww=[]
with open('/Users/huangshuhui/Google
Drive/study/cs289/hw2017/hw5/data/hw5_spam_dist/words.csv','r') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        ww.append(row)
test_tree = DecisionTree(20,len(train_x[1]))
test_tree.train(train_x,train_y)
test_tree.root.split_rule

#find most common splits in a 200-tree forest (spam) #
test_forest = RandomForest(200,800,8,20)
test_forest.train(train_x,train_y)
roots=[]
for t in test_forest.trees:
    if t.root.is_leaf()==False:
        roots.append(t.root.split_rule[0])
for rule in Counter(roots).most_common(10):
    print("("+ww[rule[0]][0]+")"+"<"+"1"+"  ("+str(rule[1])+")")

## get prediction from random forest##
final_forest = RandomForest(80,1000,25,20)
final_forest.train(train_x,train_y)
```

```
final_result = final_forest.predict(test_x)
np.savetxt('spam_predict_rf.csv', final_result, delimiter = ',')


##problem 6: census data##
###census data###
census_train=[]
census_test=[]
census_data = csv.DictReader(open('/Users/huangshuhui/Google Drive/study/cs289/hw2017/hw5/data/hw5_census_dist/train_data.csv'))
census_test_data = csv.DictReader(open('/Users/huangshuhui/Google Drive/study/cs289/hw2017/hw5/data/hw5_census_dist/test_data.csv'))
census_mode = pd.read_csv('/Users/huangshuhui/Google Drive/study/cs289/hw2017/hw5/data/hw5_census_dist/train_data.csv')

#find the mode of each category#
Mode={}
for key in census_mode.keys():
    Mode[key]=Counter(census_mode[key]).most_common(1)[0][0]

#replace missing value with the mode#
for row in census_data:
    for key in census_mode.keys():
        if row[key]=="?":
            row[key]=Mode[key]
    row['capital-gain']=float(row['capital-gain'])
    row['capital-loss']=float(row['capital-loss'])
    row['label']=int(row['label'])
    row['education-num']=float(row['education-num'])
    row['hours-per-week']=float(row['hours-per-week'])
    row['age']=float(row['age'])
    row['fnlwgt']=float(row['fnlwgt'])
    census_train.append(row)

for row in census_test_data:
    for key in census_mode.keys():
        if key!='label' and row[key]=="?":
```

```
            row[key]=Mode[key]
        row['capital-gain']=float(row['capital-gain'])
        row['capital-loss']=float(row['capital-loss'])
        row['education-num']=float(row['education-num'])
        row['hours-per-week']=float(row['hours-per-week'])
        row['age']=float(row['age'])
        row['fnlwgt']=float(row['fnlwgt'])
        census_test.append(row)

# get vectorized training labels and training data #
v_train = DictVectorizer(sparse=False)
train = v_train.fit_transform(census_train)
index_label= v_train.get_feature_names().index('label')
census_train_label=train[:,index_label]
census_train_data=train[:,[i for i in range(len(train[0])) if i!=index_label]]

# get vectorized testing data #
v_test = DictVectorizer(sparse=False)
test=v_test.fit_transform(census_test)
census_test_data=test

wgt_index=[v_test.get_feature_names().index('fnlwgt')]

##find splits in decision tree##
feature_names=v_test.get_feature_names()
test_tree_census = DecisionTree(10,len(census_train_data[0]))
test_tree_census.train(census_train_data,census_train_label)


# find most common splits in a 100-tree forest (census)#
test_forest_census = RandomForest(100,800,30,10)
test_forest_census.train(census_train_data,census_train_label)
roots_census=[]
for t in test_forest_census.trees:
    if t.root.is_leaf()==False:
        roots_census.append(t.root.split_rule)
for rule in Counter(roots_census).most_common(10):
```

```
    print("("+feature_names[rule[0][0]]+")"+"<"+str(rule[0][1])+"
("+str(rule[1])+")")

##plot validation accuracy with different depth##
accuracies=[]
length=len(census_train_data)
train_index = [x for x in range(length) if x % 5 != 0]
validation_index = [x for x in range(length) if x % 5 == 0]
cv_train_x=np.array([census_train_data[i] for i in train_index])
cv_train_y=np.array([census_train_label[i] for i in train_index])
cv_test_x=np.array([census_train_data[i] for i in validation_index])
cv_test_y=np.array([census_train_label[i] for i in validation_index])
for i in (1,20,40,60,80,100):
    tree_census = DecisionTree(i,len(census_train_data[0]))
    tree_census.train(cv_train_x,cv_train_y)
    cv_predict_y=tree_census.predict(cv_test_x)
    accuracies.append(sum(cv_predict_y==cv_test_y)/len(cv_test_y))

x=(1,20,40,60,80,100)
plt.plot(x,accuracies)
plt.xlabel('depth')
plt.ylabel('accuracy')
plt.title('accuracy with different depth')

## get prediction from random forest##
final_census_forest = RandomForest(10,2000,70,10)
final_census_forest.train(census_train_data,census_train_label)
final_census_result = final_census_forest.predict(census_test_data)
np.savetxt('census_predict_rf.csv', final_census_result, delimiter = ',')


##problem 7: Titanic dataset##
titanic_train=[]
titanic_test=[]
titanic_data = csv.DictReader(open('/Users/huangshuhui/Google
Drive/study/cs289/hw2017/hw5/data/hw5_titanic_dist/titanic_training.csv'))
```

```
titanic_test_data = csv.DictReader(open('/Users/huangshuhui/Google
Drive/study/cs289/hw2017/hw5/data/hw5_titanic_dist/titanic_testing_data.csv')
)
titanic_mode = pd.read_csv('/Users/huangshuhui/Google
Drive/study/cs289/hw2017/hw5/data/hw5_titanic_dist/titanic_training.csv')

Mode={}
for key in titanic_mode.keys():
   Mode[key]=Counter(titanic_mode[key]).most_common(1)[0][0]


### replace "" with the mode of the feature ###
### change the non-categorical variable to numeric ###
for row in titanic_data:
   for key in titanic_mode.keys():
      if row[key]=="":
         row[key]=Mode[key]
   row['survived']=float(row['survived'])
   row['pclass']=float(row['pclass'])
   row['age']=float(row['age'])
   row['sibsp']=float(row['sibsp'])
   row['parch']=float(row['parch'])
   row['fare']=float(row['fare'])
   titanic_train.append(row)
for row in titanic_test_data:
   for key in titanic_mode.keys():
      if key!='survived' and row[key]=="":
         row[key]=Mode[key]
   row['pclass']=float(row['pclass'])
   row['age']=float(row['age'])
   row['sibsp']=float(row['sibsp'])
   row['parch']=float(row['parch'])
   row['fare']=float(row['fare'])
   titanic_test.append(row)

#extract vectorized training labels and training data #
ti_train = DictVectorizer(sparse=False)
```

```
train = ti_train.fit_transform(titanic_train)
index_survived= ti_train.get_feature_names().index('survived')
titanic_train_label=train[:,index_survived]
titanic_train_data=train[:,[i for i in range(len(train[0])) if i!=index_survived]]

# get vectorized testing data #
ti_test = DictVectorizer(sparse=False)
test=ti_test.fit_transform(titanic_test)
titanic_test_data=test

feature_ti_names=ti_test.get_feature_names()

##predict the titanic testing data##
final_titanic_forest = RandomForest(20,500,10,20)
final_titanic_forest.train(titanic_train_data,titanic_train_label)
final_titanic_result = final_titanic_forest.predict(titanic_test_data)
np.savetxt('titanic_predict_rf.csv', final_titanic_result, delimiter = ',')

##test accuracies of titanic data set##
accuracies=[]
length=len(titanic_train_data)
train_index = [x for x in range(length) if x % 5 != 0]
validation_index = [x for x in range(length) if x % 5 == 0]
cv_train_x=np.array([titanic_train_data[i] for i in train_index])
cv_train_y=np.array([titanic_train_label[i] for i in train_index])
cv_test_x=np.array([titanic_train_data[i] for i in validation_index])
cv_test_y=np.array([titanic_train_label[i] for i in validation_index])
tree = DecisionTree(20,len(titanic_train_data[0]))
tree.train(cv_train_x,cv_train_y)
cv_predict_y=tree.predict(cv_test_x)
accuracies.append(sum(cv_predict_y==cv_test_y)/len(cv_test_y))
```