CS189/CS289A – Spring 2017 — Homework 5

Yaoyang Zhang, SID 3032114788

Problem 1

See code for details

```
import numpy as np
import random
import collections


class Node:
    def __init__(self, split_rule, left, right, label, is_leaf):
        self.split_rule = split_rule
        self.left = left
        self.right = right
        self.label = label
        self.is_leaf = is_leaf

    def get_split_rule(self):
        return self.split_rule

    def get_left_child(self):
        return self.left

    def get_right_child(self):
        return self.right

    def is_leaf(self):
        return self.is_leaf

    def get_label(self):
        return self.label


class DecisionTree:
    def __init__(self, max_depth, num_features, measure='entropy'):
        self.max_depth = max_depth
        self.num_features = num_features
        self.measure = measure
        self.root = Node(None, None, None, None, False)
    def cost(self, prob):
        if self.measure == 'entropy':
            if float(prob) == 0.0:
                return 0
            elif float(prob) == 1.0:
                return 0
            else:
                return - prob * np.log2(prob) - (1 - prob) * np.log2(1 - prob)
        elif self.measure == 'polynomial':
            if prob == 0.0:
                return 0
            elif prob == 1.0:
                return 0
            else:
                return (1 - prob) * prob
        else:
            print('Error. No method found')
            return None

    def impurity(self, left_label_hist, right_label_hist):
        left_size = sum(left_label_hist)
        right_size = sum(right_label_hist)
        if left_size == 0:
            right_prob = right_label_hist[0] / right_size
            right_cost = self.cost(right_prob)
            return right_cost
        elif right_size == 0:
            left_prob = left_label_hist[0] / left_size
            left_cost = self.cost(left_prob)
            return left_cost
        else:
            left_prob = left_label_hist[0] / left_size
            left_cost = self.cost(left_prob)
            right_prob = right_label_hist[0] / right_size
            right_cost = self.cost(right_prob)
            return (left_size * left_cost + right_size * right_cost) / (left_siz



    def segmenter(self, data, labels):
        threshold_list = (np.mean(data[labels == 1], axis=0) + np.mean(data[labe
        feature_collection = random.sample(range(len(data[0])), self.num_feature
        impurity_list = []
        for i in feature_collection:
            threshold = threshold_list[i]
            group1_labels = labels[data[:, i] >= threshold]
            group2_labels = labels[data[:, i] < threshold]
            left_labels_hist = [len(group1_labels) - sum(group1_labels), sum(gro
            right_labels_hist = [len(group2_labels) - sum(group2_labels), sum(gr
            impurity_list.append(self.impurity(left_labels_hist, right_labels_hi
        best_feature_index = feature_collection[impurity_list.index(min(impurit
        best_feature_threshold = threshold_list[best_feature_index]
        # print('best feature is: ',best_feature_index, ', best threshold is: ',
        return best_feature_index, best_feature_threshold


    def grow_tree(self, data, labels, height):
        # print('depth of this node: ',height)
        if height <= self.max_depth:
            if len(np.unique(labels)) == 1:
                # print('pure leaf: ',np.unique(labels)[0])
                return Node(None, None, None, np.unique(labels)[0], True)
            # elif len(labels) < 10:
            #     common_label = collections.Counter(labels).most_common(1)[0][0
            #     return Node(None, None, None, common_label, True)
            else:
                [index, threshold] = self.segmenter(data, labels)
                left = data[:, index] < threshold
                right = data[:, index] >= threshold
                left_data = data[left]
                right_data = data[right]
                left_label = labels[left]
                right_label = labels[right]
                if len(left_label) == 0 or len(right_label) == 0:
                    common_label = collections.Counter(labels).most_common(1)[0]
                    # print('none leaf: ', common_label)
                    return Node(None, None, None, common_label, True)
                else:
                    left_node = self.grow_tree(left_data, left_label, height + 1
                    right_node = self.grow_tree(right_data, right_label, height
                    return Node((index, threshold), left_node, right_node, None,

        else:
            common_label = collections.Counter(labels).most_common(1)[0][0]
            # print('reach max: ',common_label)
            return Node(None, None, None, common_label, True)


    def traverse(self, root, x):
        if root.is_leaf == True:
            print('reach leaf', root.label)
            return root.label
        else:
            index = root.split_rule[0]
            threshold = root.split_rule[1]
            if x[index] <= threshold:
                print('feature: ',index, ' <= ',threshold )
                return self.traverse(root.left, x)
            else:
                print('feature: ', index, ' >', threshold)
                return self.traverse(root.right, x)


    def train(self, data, labels):
        self.root = self.grow_tree(data, labels, 1)
        print(self.root.split_rule)

    def predict(self, data):
        predict_res = []
        for x in data:
            predict_res.append(self.traverse(self.root, x))
        return predict_res
```

Problem 2

```
from decisionTree import DecisionTree
import numpy as np


class RandomForest:
    def __init__(self, num_trees, max_depth, num_sample, num_feature):
        self.num_trees = num_trees
        self.max_depth = max_depth
        self.num_sample = num_sample
        self.num_feature = num_feature
        self.trees = []

    def train(self, data, labels):
        self.data = data
        self.labels = labels
        for i in range(self.num_trees):
            sample_index = np.random.choice(self.data.shape[0], self.num_sample,
            train_data = self.data[sample_index, :]
            train_labels = self.labels[sample_index]
            tree = DecisionTree(self.max_depth,self.num_feature)
            tree.train(train_data, train_labels)
            self.trees.append(tree)


    def predict(self, data):
        labels = []
        for tree in self.trees:
            labels.append(tree.predict(data))
        res = np.mean(np.array(labels),axis=0)
        for i in range(len(res)):
            if res[i] >= 0.5:
                res[i] = 1
            else:
                res[i] = 0
        return res
```

Problem 3

(a) For categorical data, I implemented "one-hot encoding". For missing data, I used the most frequent value in that feature (for census) or the mean value in that feature (for titanic) to represent the missing value.

(b) My stopping criteria are (1) there is only one label in the subset (2) the size of the subset is 1 (3) one of the split subsets has size 0.

(c) To speed up training process, for each feature x_i, I use the average of the mean values of x_is that belong to class 1 and the mean values of x_is that belong to class 0 as the threshold x_s. That is $x_s = \frac{1}{2}(\frac{1}{|C_0|}\sum_{i \in C_0} x_i + \frac{1}{|C_1|}\sum_{j \in C_1} x_j)$. In this case, it takes $O(dh)$ time to train a decision tree where d is the size of dimension and h is the height of the tree.

(d) For each decision tree, we only feed a random subset of the training data to it (equal size of the original training set but may contain duplicates), and at each split, it can only split on a random subset of the feature space. Other hyper-parameters include the maximum depth of the tree and the number of trees. We use the mean value of the predictions from all the trees as the final result.

Problem 4

(a) Training and validation accuracy are shown as follows (with no extra features, maximum depth = 10, number of decision trees = 10)

		Training Accuracy	Validation Accuracy
Spam	Decision Tree	0.780	0.766
	Random Forest	0.766	0.747
Census	Decision Tree	0.8533	0.8568
	Random Forest	0.8567	0.8707
Titanic	Decision Tree	0.904	0.783
	Random Forest	0.916	0.76

(b) Kaggle name: Yaoyang Zhang
Spam score: 0.95240
Census score: 0.85358
Titanic score: 0.78710

Problem 5

(a) I used the bag-of-words model and extract the top 100 frequent words from both spam and ham emails and use the counts of these words as features. See details in featurize.py.

(b) For a decision tree with depth 4 (with no extra features):
feature: 28 ('!') $\leq$ 1.23842021577
feature: 28 ('!') $\leq$ 0.205145957074
feature: 3 ('money') $\leq$ 0.176469983345
feature: 29 ('(') $\leq$ 1.01540267221
It is not a spam.

(c) For a random forest with 30 trees and 10 (out of 32) features available at each split, the top 4 most common splits and their counts at root are:
'!': 8
'energy': 6
'money': 6
'prescription': 4

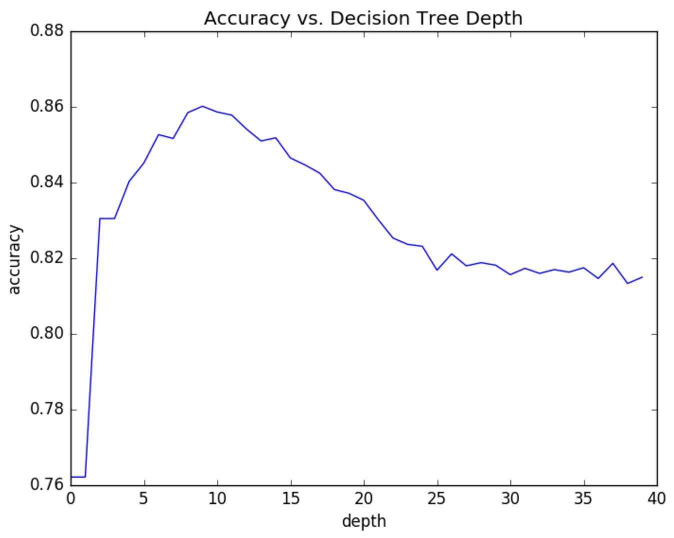
Problem 6

(a) I did not use any features other than ”one-hot encoding”.

(b) For a decision tree with depth 4:
feature: 25 (’widowed’) $\leq$ 0.595353702345
feature: 102 (’capital gain’) $\leq$ 3432.16711525
feature: 101 (’education number’) $\leq$ 10.9111214735
feature: 99 (’age’) $>$ 39.253670414
Salary is less than 50000 (label =0).

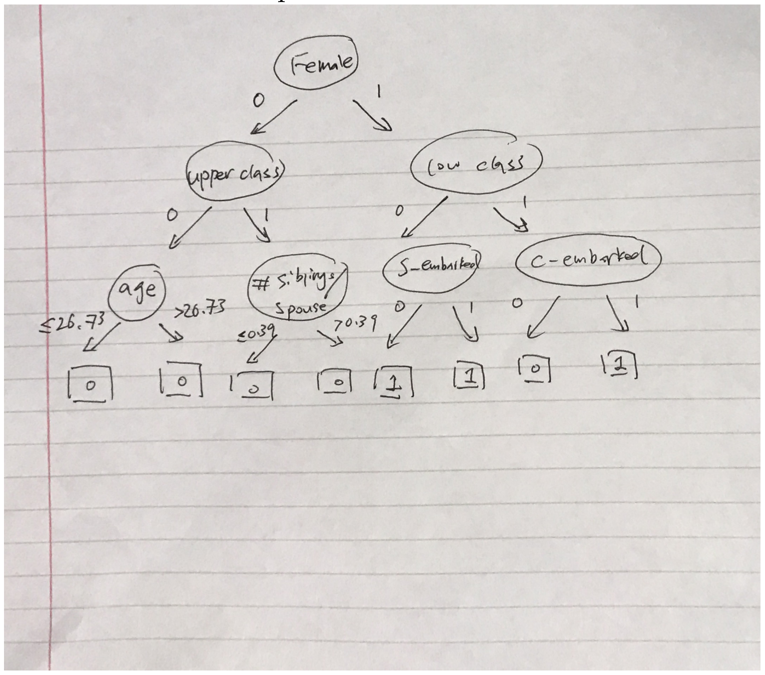
(c) For a random forest with 30 trees and 30 (out of 105) features available at each split, the top 4 most common splits and their counts at root are:
’Married-spouse-absent’: 11
’Widowed’: 7
’Wife’: 4
’Education number’: 2

(d) The plot is as follows. The best number of depth is 10. After 10, the validation accuracy begins to drop because the decision tree is trying to overfit the training data.



Problem 7

Visualization for a depth-3 decision tree is as follows


Female
0
1
upper class
low class
age
≤26.73
>26.73
# Siblings spouse
≤0.39
>0.39
S-embarked
C-embarked
0
1
0
0
0
0
1
1
0
1