CS189/289A – Spring 2017 — Homework 5

Yicheng Chen, SID 26943685

Collaborators: NONE)

1. Implement decision trees

Code:

```
class decision_tree:

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

    def __init__(self, max_depth=1e10):
        self.max_depth = max_depth

    def max_count(self, array):
        return max(set(array), key=array.tolist().count)

    # utility function for entropy calculation
    def entropy(self, indices):
        p = itemfreq(a)[:, 1].astype(float) / len(indices)
        return -p.dot(np.log2(p))

    # calculate entropy the number of instances in each class in known
    def entropy_n(self, all_n):
        p = all_n / (np.sum(all_n)+1e-20)
        return -p.dot(np.log2(p+1e-20))

    # calculate the impurity("badness") of the specified split on the input data
    def impurity(self, left_label_hist, right_label_hist):
        Sl = np.sum(left_label_hist)
        Sr = np.sum(right_label_hist)
        return (Sl*self.entropy_n(left_label_hist) + Sr * self.entropy_n(right_label_hist))

    # find the threshold that best split the data points with a certain feature
    # Note: <= th goes to S_left and > th goes to S_right
    def find_threshold(self, feature, labels):
        all_f = sorted(set(feature)) # sorted in ascending order
        all_l = set(labels) # list unique labels

        freq_mat = np.zeros([len(all_f), len(all_l)])
        for i, f in enumerate(all_f):
            for j, l in enumerate(all_l):
                freq_mat[i, j] = len(labels[np.where(labels[np.where(feature==f)]==l)])

        # calculate the average of two neighboring values as threshold
        # iterates from min to max
        all_threshold = (np.hstack((all_f[1:], all_f[-1])) + all_f) / 2.

        # in the beginning, all goes to the right node
        n_left = np.zeros([len(all_l)])
        n_right = np.sum(freq_mat, axis=0)
        n_left_sum = 0
        min_threshold = all_threshold[0]
        min_H = self.impurity(n_left, n_right)
        # loop through all threshold to find the one with the minimum impurity
        for i, th in enumerate(all_threshold):
            n_left += freq_mat[i, :]
            n_right -= freq_mat[i, :]
            H = self.impurity(n_left, n_right)
            if H < min_H:
                min_H = H
                min_threshold = th
        return min_threshold, min_H


    # find the best feature and threshold to split data points
    def segmenter(self, data, labels, m=-1):
        d = data.shape[1]
        if m == -1:
            all_features = np.arange(d)
        else:
            all_features = np.random.choice(range(d), m, replace=False)
        min_H = 1e20
        min_th = 0
        min_i = 0
        for i in all_features:
            threshold, H = self.find_threshold(data[:, i], labels)
            if H < min_H:
                min_H = H
                min_th = threshold
                min_i = i
        return min_i, min_th
    # the recurrence function that builds the decision tree
    def grow_tree(self, S, depth, m=-1):
        if len(set(self.labels[S])) == 1 or depth >= self.max_depth: # pure node or reach m
            return self.node(left=None, right=None, split_rule=None, is_leaf=1, \
                             label=self.max_count(self.labels[S]))
        else:
            min_i, min_th = self.segmenter(self.data[S, :], self.labels[S], m=m)
            # the following comprehension might be slow
#             Sl = [j for j, x in enumerate(self.data[:, min_i]) if x <= min_th and j in S]
#             Sr = [j for j, x in enumerate(self.data[:, min_i]) if x > min_th and j in S]
            # update: faster:
            Sl = [j for j in S if self.data[j, min_i] <= min_th]
            Sr = [j for j in S if self.data[j, min_i] > min_th]
            # another: faster:
#             Sl = np.intersect1d(np.where(self.data[:, min_i] <= min_th), S)
#             Sr = np.intersect1d(np.where(self.data[:, min_i] > min_th), S)
            if len(Sl) == 0 or len(Sr) == 0:
                return self.node(left=None, right=None, split_rule=None, is_leaf=1, \
                                 label=self.max_count(self.labels[S]))
            else:
                return self.node(left=self.grow_tree(Sl, depth+1, m), right=self.grow_tree(
                                 split_rule = (min_i, min_th), \
                            is_leaf=0, label=None)

    # train the decision tree
    def train(self, data, labels, m=-1):
        self.data = data
        self.labels = labels
        S = np.array(range(len(labels)))
        self.root = self.grow_tree(S, 1, m=m)
        self.data = None
        self.labels = None
        return self

    # predict labels of test data
    def predict(self, data, verbose=False):
        if data.ndim == 1: # special case of only 1 row (it becomes a 1d vector in numpy)
            data = np.reshape(data, [1, len(data)])
            N = 1
        else:
            N = data.shape[0]
        labels = np.zeros(N)

        # predict each data point
        for i in range(N):
            d = data[i, :]
            current_node = self.root
            # going down along the tree
            while not current_node.is_leaf: # not reach leaf yet
                idx = current_node.split_rule[0]
                th = current_node.split_rule[1]
                if d[idx] <= th:
                    current_node = current_node.left
                    if verbose:
                        print('Going left')
                else:
                    current_node = current_node.right
                    if verbose:
                        print('Going right')
            if verbose:
                print()

            labels[i] = current_node.label
        return labels

    # calculate the prediction accuracy
    def accuracy(self, data, true_labels):
        labels = self.predict(data, verbose=False)
        N = len(labels)
        return np.sum(labels == true_labels) / float(N)
```

2. Implement random forests

Code:

```
class random_forest:
    def __init__(self, n_trees=20, n_sample=1000, n_feature=-1, max_depth=1e10):
        self.n_trees = n_trees
        self.n_sample = n_sample
        self.n_feature = -1
        self.max_depth = max_depth
        self.trees = np.array([decision_tree(max_depth)] * n_trees)

    def train_parallel(self, data, labels, verbose=True):
        if self.n_feature == -1:
            d = data.shape[1]
            self.n_feature = int(np.sqrt(d)) # num of random features = sqrt(d) is a good g
        pool = mp.Pool()
        results = np.zeros(self.n_trees, dtype=object)
        print('# Trees = %d' % self.n_trees)
        for i, dt in enumerate(self.trees):
#             print(i)
            idx = np.random.choice(range(len(data)), self.n_sample)
            sub_data = data[idx, :]
            sub_labels = labels[idx]
            results[i] = pool.apply_async(dt.train, args=(sub_data, sub_labels, self.n_feat
            if verbose:
                print('%d: Tree loaded.' % i)
#             dt.train(sub_data, sub_labels, m=self.n_feature) # activate the random featur
        self.trees = [p.get() for p in results]
        print('Trees trained.')
        pool.close()

    def train(self, data, labels):
        if self.n_feature == -1:
            d = data.shape[1]
            self.n_feature = int(np.sqrt(d)) # num of random features = sqrt(d) is a good g
        print('# Trees = %d' % self.n_trees)
        for i, dt in enumerate(self.trees):
            print(i)
            idx = np.random.choice(range(len(data)), self.n_sample)
            sub_data = data[idx, :]
            sub_labels = labels[idx]
            dt.train(sub_data, sub_labels, m=self.n_feature) # activate the random feature 

    def predict(self, data, verbose=False):
        if data.ndim == 1: # special case of only 1 row (it becomes a 1d vector in numpy)
            data = np.reshape(data, [1, len(data)])
            N = 1
        else:
            N = data.shape[0]
        labels = np.zeros(N)

        # predict each data point
        for i in range(N):
            votes = np.zeros(self.n_trees)
            for j, t in enumerate(self.trees):
                votes[j] = t.predict(data[i, :], verbose=verbose)
            labels[i] = t.max_count(votes)
        return labels

    def accuracy(self, data, true_labels):
        labels = self.predict(data, verbose=False)
        N = len(labels)
        return np.sum(labels == true_labels) / float(N)
```

3. Describe implementation details

(a) Categorical features are vectorized. Missing values are replaced by the most frequent element or mean value.

(b) The stopping criteria is the maximum depth of tree.

(c) Nothing special to speed up decision tree. But for random forest, the multiprocessing module is used to train trees in parallel.

(d) The random forest is basically a collection of decision trees with random data sampling and feature sampling. Only a wrapper class and small modification of the decision tree class are needed.

(e) I think the parallel part is quite cool!

4. Performance evaluation

(a) Decision tree:

Data	Training Accuracies	Validation Accuracies
Spam	0.8071	0.7848
Census	0.8924	0.8400
Titanic	0.9237	0.77

Random forest:

Data	Training Accuracies	Validation Accuracies
Spam	0.7979	0.7816
Census	0.8924	0.8592
Titanic	0.9237	0.87

(b) Kaggle:
Display name: Yicheng Chen
Public scores:

Data	Public score
Spam	0.97880
Census	0.86450
Titanic	0.81290

5. Writeup for the spam dataset

(a) Bag-of-words and TF-IDF features are used for the spam dataset.

(b) The first data point of the training set (max depth=15):

```
Feature 28 <= 0.500000
Feature 29 <= 0.500000
Feature 19 <= 0.500000
Feature 25 <= 1.500000
Feature 7 <= 0.500000
Feature 13 <= 0.500000
Feature 3 <= 0.500000
Feature 0 <= 0.500000
Feature 26 <= 0.500000
Feature 31 <= 0.500000
Feature 6 <= 0.500000
Feature 20 <= 0.500000
Feature 16 <= 0.500000
Feature 30 <= 0.500000
Classification: 0
```

(c) Random forest split at root (20 trees):

```
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 16 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
Feature 19 <= 0.500000
```

Feature 19 <= 0.5 (19 trees)
Feature 16 <= 0.5 (1 tree)

6. Writeup for the census dataset

(a) The 2nd order polynomial features are used.

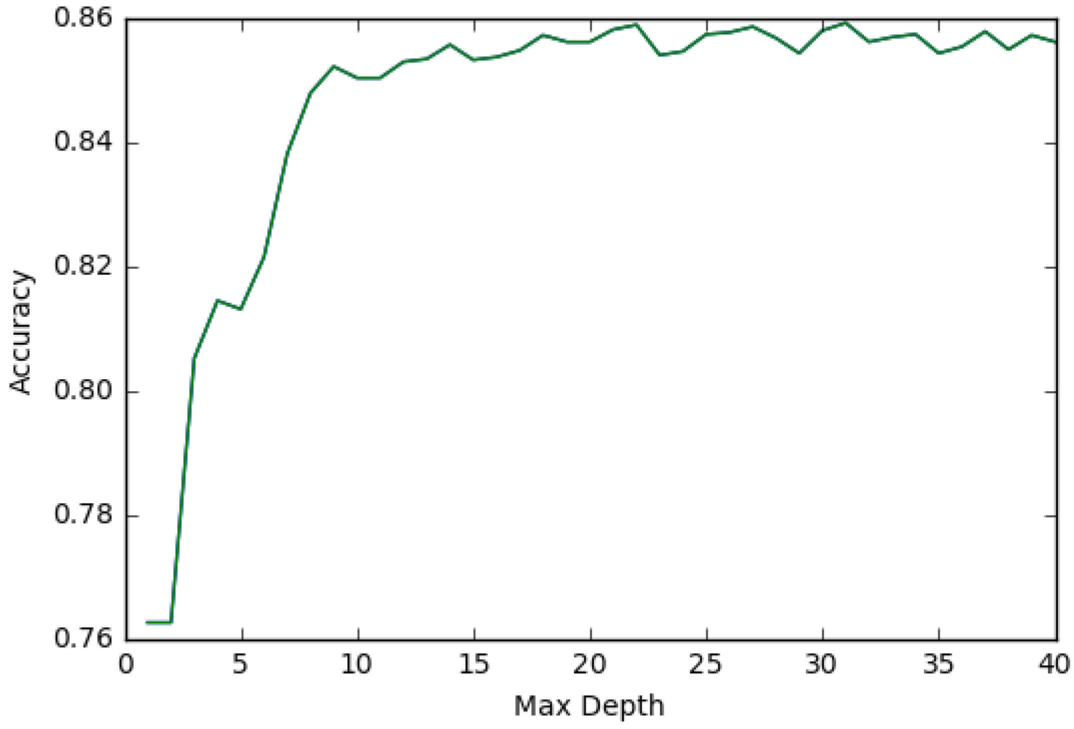
(b) The first data point of the training set (max depth=15):

```
Feature 54 <= 0.500000
Feature 4 <= 7055.500000
Feature 21 <= 488.000000
Feature 12 > 1332.500000
Feature 5 <= 2218.500000
Feature 84 > 0.500000
Feature 9 <= 394.000000
Feature 1 > 32.500000
Feature 8 > 11725338.500000
Classification: 0
```

(c) Random forest split at root (20 trees):

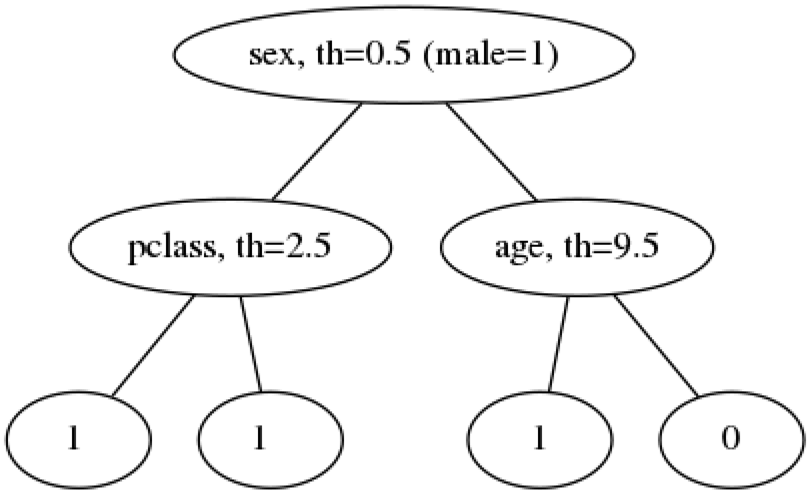
```
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
Feature 48 <= 0.500000
```

(d) The accuracy starts low because the bias is high when the trees are limited in depth. But it then rises because the bias is reduced with deeper trees.


Accuracy
0.86
0.84
0.82
0.80
0.78
0.76
0
5
10
15
20
25
30
35
40
Max Depth

7. Writeup for the Titanic dataset

The tree looks like this:


sex, th=0.5 (male=1)
pclass, th=2.5
age, th=9.5
1
1
1
0