```
# You may want to install "gprof2dot"
import io
from collections import Counter

import numpy as np
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin

from subprocess import check_call

import pydot

eps = 1e-5  # a small number)
def split(X,Y,thresh):
        idx0 = np.where(X < thresh)[0]
        idx1 = np.where(X >= thresh)[0]
        #print(sum(idx0))
        #print(sum(idx1))
        X0, X1 = X[idx0], X[idx1]
        y0 ,y1 = y[idx0], y[idx1]
        return X0,y0,X1,y1


class DecisionTree:
    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None  # for non-leaf nodes
        self.split_idx, self.thresh = None, None  # for non-leaf nodes
        self.data, self.pred = None, None  # for leaf nodes

    @staticmethod
    def information_gain(X, y, thresh):
        X0,y0,X1,y1 = split(X,y, thresh)
        def g(y):
            pyk_2 = 0
            for i in range(2):
                try:
                    pyeqk = len(y[y == i])/float(len(y))
                    pyk_2 = pyeqk**2
                except ZeroDivisionError:
                    pyeqk = 0
            return 1 - pyk_2

        return len(X0)*g(y0)/float(len(X)) + len(X1)*g(y1)/float(len(X))

    @staticmethod
    def gini_impurity(X, y, thresh):
        X0,y0,X1,y1 = split(X,y, thresh)

        def h(y) :
            hy = 0
            for i in range(2) :
                try :
                    pyeqk = len(y[y == i])/float(len(y))
                    hy = hy - pyeqk*np.log(pyeqk)
                except ZeroDivisionError :
                    pyeqk = 0
```

```
            return hy

        return len(X0)*h(y0)/float(len(X)) + len(X1)*h(y1)/float(len(X))



    def split(self, X, y, idx, thresh):
        X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
        y0, y1 = y[idx0], y[idx1]
        return X0, y0, X1, y1

    def split_test(self, X, idx, thresh):
        idx0 = np.where(X[:, idx] < thresh)[0]
        idx1 = np.where(X[:, idx] >= thresh)[0]
        X0, X1 = X[idx0, :], X[idx1, :]
        return X0, idx0, X1, idx1

    def fit(self, X, y):
        if self.max_depth > 0:
            # compute entropy gain for all single-dimension splits,
            # thresholding with a linear interpolation of 10 values
            gains = []
            # The following logic prevents thresholding on exactly the minimum
            # or maximum values, which may not lead to any meaningful node
            # splits.
            thresh = np.array([
                np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
                for i in range(X.shape[1])
            ])
            for i in range(X.shape[1]):
                gains.append([self.information_gain(X[:, i], y, t) for t in thresh
[i, :]])

            gains = np.nan_to_num(np.array(gains))
            self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains),
gains.shape)
            self.thresh = thresh[self.split_idx, thresh_idx]
            X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx, thresh=self.thresh)
            if X0.size > 0 and X1.size > 0:
                self.left = DecisionTree(
                    max_depth=self.max_depth - 1, feature_labels=self.features)
                self.left.fit(X0, y0)
                self.right = DecisionTree(
                    max_depth=self.max_depth - 1, feature_labels=self.features)
                self.right.fit(X1, y1)
            else:
                self.max_depth = 0
                self.data, self.labels = X, y
                self.pred = stats.mode(y).mode[0]
        else:
            self.data, self.labels = X, y
            self.pred = stats.mode(y).mode[0]
        return self

    def predict(self, X):
        if self.max_depth == 0:
            return self.pred * np.ones(X.shape[0])
        else:
            X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx,
thresh=self.thresh)
            yhat = np.zeros(X.shape[0])
```

```
            yhat[idx0] = self.left.predict(X0)
            yhat[idx1] = self.right.predict(X1)
            return yhat


class BaggedTrees(BaseEstimator, ClassifierMixin):
    def __init__(self, params=None, n=200):

        if params is None:
            params = {}
        self.params = params
        self.n = n
        self.decision_trees = [
            sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
            for i in range(self.n)
        ]

    def fit(self, X, y):
        self.mask = []
        for tree in self.decision_trees:
                mask = np.random.randint(0, high = len(X), size= len(X))
                Xsampling = X[mask,:]
                ysampling = y[mask]
                tree.fit(Xsampling,ysampling)
                self.mask.append(mask)



    def predict(self, X):
        preds = []
        for tree in self.decision_trees:
                preds.append(tree.predict(X))
        return stats.mode(np.array(preds), axis = 0 )[0].reshape(len(X))

class RandomForest(BaggedTrees):
    def __init__(self, params=None, n=200, m=2):
        super().__init__(params = params , n = n )
        self.m = m

    def fit(self, X,y):
        self.mask = []
        self.features = []
        for tree in self.decision_trees:
                mask = np.random.randint(0, high = len(X), size= len(X))
                features = np.random.choice( X.shape[1], size = self.m )
                Xsampling = X[mask,:]
                Xsampling = Xsampling[:,features]
                ysampling = y[mask]
                tree.fit(Xsampling,ysampling)
                self.mask.append(mask)
                self.features.append(features)
    def predict(self,X):
        preds = []
        k = 0
        for tree in self.decision_trees:
                preds.append(tree.predict(X[:,self.features[k]]))
        return stats.mode(np.array(preds), axis = 0 )[0].reshape(len(X))



def checkXY(x,y,tree):
```

```
        ypred = tree.predict(x.reshape(1,-1))
        if ypred == y :
                return 1
        else :
                return 0

class BoostedRandomForest(RandomForest):
    def fit(self, X, y ):
        self.w = np.ones(X.shape[0]) / X.shape[0]  # Weights on data
        self.a = np.zeros(self.n)  # Weights on decision trees
        k = 0
        self.features = []
        for tree in self.decision_trees:
                mask = np.random.randint(0, high = len(X), size= len(X))
                features = np.random.choice(X.shape[1], size = self.m )
                Xsampling = X[mask,:]
                Xsampling = Xsampling[:,features]
                ysampling = y[mask]
                tree.fit(Xsampling,ysampling)
                self.features.append(features)

                ej = 0
                for j in range(len(Xsampling)):
                        ej = checkXY(Xsampling[j,:], ysampling[j], tree )*self.w[j] +
ej

                ej = ej/float(sum(self.w))

                self.a[k] = 0.5*np.log((1-ej)/float(ej))

                for i in range(len(Xsampling)):
                        if checkXY(Xsampling[i,:],ysampling[i], tree) > 0.5 :
                                self.w[i] = self.w[i]*np.exp(self.a[k])
                        else :
                                self.w[i] = self.w[i]*np.exp(-self.a[k])
                k = k + 1


    def predict(self, X):
        classes = list(set(y))
        preds_tot = []
        for i in range(len(X)):
                preds = []
                for c in classes :
                        zj = 0
                        k = 0
                        for tree in self.decision_trees:
                                Xcheck = X[:,self.features[k]]
                                Xcheck = Xcheck[i,:]
                                zj = zj + self.a[k]*checkXY(Xcheck, c, tree)
                                k = k + 1
                        preds.append(zj)
                preds_tot.append(classes[np.argmax(preds)])
        return preds_tot


def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
    # fill_mode = False

    # Temporarily assign -1 to missing data
    data[data == b''] = '-1'
```

```
    # Hash the columns (used for handling strings)
    onehot_encoding = []
    onehot_features = []
    for col in onehot_cols:
        counter = Counter(data[:, col])
        for term in counter.most_common():
            if term[0] == b'-1':
                continue
            if term[-1] <= min_freq:
                break
            onehot_features.append(term[0])
            onehot_encoding.append((data[:, col] == term[0]).astype(np.float))
        data[:, col] = '0'
    onehot_encoding = np.array(onehot_encoding).T
    data = np.hstack([np.array(data, dtype=np.float), np.array(onehot_encoding)])

    # Replace missing data with the mode value. We use the mode instead of
    # the mean or median because this makes more sense for categorical
    # features such as gender or cabin type, which are not ordered.
    if fill_mode:
        for i in range(data.shape[-1]):
            mode = stats.mode(data[((data[:, i] < -1 - eps) +
                                    (data[:, i] > -1 + eps))][:, i]).mode[0]
            data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode

    return data, onehot_features

def mostcommon(lis):
    reps = []
    for i in list(set(lis)):
        reps.append(sum(i == np.array(lis)))
    return lis[np.argmax(reps)]

def evaluate(clf):
    print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
    if hasattr(clf, "decision_trees"):
        counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
        first_splits = [(features[term[0]], term[1]) for term in counter.most_common
()]
        print("First splits", first_splits)
def get_acc(X,y,model):
        preds = model.predict(X)
        right = np.array(np.equal(y,preds)).astype(int)

        return sum(right)/float(len(y))

def kfold_3 (X,y, model):
        acc = []
        acc_train = []
        for f in range(1,4):
                inds = np.linspace(0,len(X)-1 , len(X)).astype(int)
                bool_test = np.logical_and(inds >((f-1) /float(3))*len(X), inds <(f /
float(3))*len(X))

                Xtrain = X[~bool_test,:]
                Xtest = X[bool_test,:]

                ytrain = y[~ bool_test]
                ytest =y[bool_test]

                model.fit(Xtrain,ytrain)
                acc_train.append(get_acc(Xtrain,ytrain,model))
```

```
                acc.append(get_acc(Xtest,ytest, model))

        acc.append(np.average(acc_train))
        return acc

if __name__ == "__main__":
    parta = False
    partd = False
    partg_a = False
    partg_b = False
    parti = False
    partj_a = True
    partj_b = False

    if parta :
        dataset = "titanic"
        params = {
          "max_depth": 5,
         # "random_state": 6,
         "min_samples_leaf": 10,
                 }

    N = 100

    if partd or partg_b or partj_a :
                dataset = "titanic"

    if partg_a or parti  :
                dataset = "spam"


    if dataset == "titanic":
        # Load titanic data
        path_train = 'titanic_training.csv'
        data = genfromtxt(path_train, delimiter=',', dtype=None)
        path_test = 'titanic_testing_data.csv'
        test_data = genfromtxt(path_test, delimiter=',', dtype=None)
        y = data[1:, 0]  # label = survived
        class_names = ["Died", "Survived"]

        labeled_idx = np.where(y != b'')[0]
        y = np.array(y[labeled_idx], dtype=np.int)
        print("\n\nPart (b): preprocessing the titanic dataset")
        X, onehot_features = preprocess(data[1:, 1:], onehot_cols=[1, 5, 7, 8])
        X = X[labeled_idx, :]
        Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
        assert X.shape[1] == Z.shape[1]
        features = list(data[0, 1:]) + onehot_features

    elif dataset == "spam":
        features = [
            "pain", "private", "bank", "money", "drug", "spam", "prescription",
"creative",
            "height", "featured", "differ", "width", "other", "energy", "business",
"message",
            "volumes", "revision", "path", "meter", "memo", "planning", "pleased",
"record", "out",
            "semicolon", "dollar", "sharp", "exclamation", "parenthesis",
"square_bracket",
            "ampersand"
        ]
        assert len(features) == 32
```

```
        # Load spam data
        path_train = 'spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]

    else:
        raise NotImplementedError("Dataset %s not handled" % dataset)

    print("Features:", features)
    print("Train/test size:", X.shape, Z.shape)

    print("\n\nPart 0: constant classifier")
    print("Accuracy", 1 - np.sum(y) / y.size)

    if parta :
    # Basic decision tree
        print("\n\nPart (a-b): simplified decision tree")
        dt = DecisionTree(max_depth=3, feature_labels=features)
        dt.fit(X, y)
        print("Predictions", dt.predict(Z)[:100])

        print("\n\nPart (c): sklearn's decision tree")
        clf = sklearn.tree.DecisionTreeClassifier(random_state=0, **params)

        clf.fit(X, y)
        evaluate(clf)
        out="tree.dot"
        sklearn.tree.export_graphviz(clf, out_file=out, feature_names=features,
class_names=class_names)

        check_call(["/usr/bin/dot","-Tpng','tree.dot','-o','tree_part_c.png'])


    if partd :

        print("doing BaggedTrees")
        btree = BaggedTrees()
        btree.fit(X,y)

        #find most
        rootfeatures = []
        root_thresh = []
        for tree in btree.decision_trees:
                rootfeatures.append(features[tree.tree_.feature[0]])
                root_thresh.append(tree.tree_.threshold[0])

    if partg_a or partg_b:
        print("Doing Random Forest")
        rf= RandomForest(m = 4 )
        rf.fit(X,y)

        #find most
        rootfeatures = []
        root_thresh = []
        for tree in rf.decision_trees:
                rootfeatures.append(features[tree.tree_.feature[0]])
                root_thresh.append(tree.tree_.threshold[0])
```

```
if parti :
    print('Doing BoostedRandomForest')
    bt = BoostedRandomForest()
    bt.fit(X,y)

if partj_a :
    print('Titanic')
    print('')
    print('DecisionTree')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,DecisionTree()))

    print('BaggedTrees')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,BaggedTrees()))


    print('RandomForest')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,RandomForest(n = 400, m = 7)))

    # print('Adaboost')
    # print('error = [kfold1, kfold2, kfold3, avg_training]')
    # print(kfold_3(X,y,BoostedRandomForest(n = 200, m = 7)))

    bt = BaggedTrees(n = 200 )
    bt.fit(X,y)
    np.savetxt('submission_t.txt',bt.predict(X).astype(int))

if partj_b :
    print('')
    print('SPAM')
    print('')
    print('DecisionTree')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,DecisionTree()))

    print('BaggedTrees')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,BaggedTrees(n = 200)))


    print('RandomForest')
    print('error = [kfold1, kfold2, kfold3, avg_training]')
    print(kfold_3(X,y,RandomForest(n = 200, m = 7)))

    # print('Adaboost')
    # print('error = [kfold1, kfold2, kfold3, avg_training]')
    # print(kfold_3(X,y,BoostedRandomForest(n = 200, m = 7)))


    bt = BaggedTrees(n = 200 )
    bt.fit(X,y)
    np.savetxt('submission_s.txt',bt.predict(X).astype(int))
```