```
"""
Author      : Jackson Crewe and Matt Guillory
Assignment  : 2
Class       : HMC CS 158
Date        : 2018 Aug 11
Description : Decision Tree Classifier
"""

# Use only the provided packages!
import collections
from util import *

# numpy libraries
import numpy as np

# scikit-learn libraries
from sklearn import tree

######################################################################
# classes
######################################################################

class Tree(object) :
    """
    Array-based representation of a binary decision tree.

    See tree._tree.Tree (a Python wrapper around a C class).
    The binary tree is represented as a number of parallel arrays. The i-th
    element of each array holds information about the node 'i'. Node 0 is the
    tree's root. NOTE: Some of the arrays only apply to either leaves or split
    nodes, resp. In this case the values of nodes of the other type are
    arbitrary!

    Attributes
    --------------------
      node_count : int
          The number of nodes (internal nodes + leaves) in the tree.

      children_left : array of int, shape [node_count]
          children_left[i] holds the node id of the left child of node i.
          For leaves, children_left[i] == TREE_LEAF. Otherwise,
          children_left[i] > i. This child handles the case where
          X[:, feature[i]] <= threshold[i].

      children_right : array of int, shape [node_count]
          children_right[i] holds the node id of the right child of node i.
          For leaves, children_right[i] == TREE_LEAF. Otherwise,
          children_right[i] > i. This child handles the case where
          X[:, feature[i]] > threshold[i].

      feature : array of int, shape [node_count]
          feature[i] holds the feature to split on, for the internal node i.

      threshold : array of double, shape [node_count]
          threshold[i] holds the threshold for the internal node i.

      value : array of double, shape [node_count, 1, max_n_classes]
          value[i][0] holds the counts of each class reaching node i

      impurity : array of double, shape [node_count]
          impurity[i] holds the impurity at node i.

      n_node_samples : array of int, shape [node_count]
          n_node_samples[i] holds the number of training samples reaching node i.
    """
    TREE_LEAF = tree._tree.TREE_LEAF
    TREE_UNDEFINED = tree._tree.TREE_UNDEFINED

    def __init__(self, n_features, n_classes, n_outputs=1) :
        if n_outputs != 1 :
            raise NotImplementedError("each sample must have a single label")

        self.n_features     = n_features
```

```
        self.n_classes      = n_classes
        self.n_outputs      = n_outputs

        capacity = 2047 # arbitrary, allows max_depth = 10
        self.node_count     = capacity
        self.children_left  = np.empty(self.node_count, dtype=int)
        self.children_right = np.empty(self.node_count, dtype=int)
        self.feature        = np.empty(self.node_count, dtype=int)
        self.threshold      = np.empty(self.node_count)
        self.value          = np.empty((self.node_count, n_outputs, n_classes))
        self.impurity       = np.empty(self.node_count)
        self.n_node_samples = np.empty(self.node_count, dtype=int)

        # private
        self._next_node     = 1 # start at root
        self._classes       = None

    #========================================
    # helper functions

    def _get_value(self, y) :
        """
        Get count of each class.

        Parameters
        --------------------
            y     -- numpy array of shape (n,), target classes

        Returns
        --------------------
            value -- numpy array of shape (n_classes,), class counts
                     value[i] holds count of each class
        """
        if len(y) == 0 :
            raise Exception("cannot separate empty set")

        counter = collections.defaultdict(lambda: 0)
        for label in y :
            counter[label] += 1

        value = np.empty((self.n_classes,))
        for i, label in enumerate(self._classes) :
            value[i] = counter[label]

        return value

    def _entropy(self, y) :
        """
        Compute entropy.

        Parameters
        --------------------
            y -- numpy array of shape (n,), target classes

        Returns
        --------------------
            H -- entropy
        """

        # compute counts
        _, counts = np.unique(y, return_counts=True)

        totalSum = 0
        H = 0
        for outcomeCount in counts:
            totalSum += outcomeCount

        for outcomeCount in counts:
            H += (-1.0) * (float(outcomeCount)/float(totalSum)) * np.log2(float(outc
omeCount)/float(totalSum))

        return H
```

```
    def _information_gain(self, Xj, y) :
        """
        Compute information gain.

        Parameters
        --------------------
            Xj             -- numpy array of shape (n,), samples (one feature only)
            y              -- numpy array of shape (n,), target classes

        Returns
        --------------------
            info_gain      -- float, information gain using best threshold
            best_threshold -- float, threshold with best information gain
        """
        n = len(Xj)
        if n != len(y) :
            raise Exception("feature vector and class vector must have same length")

        # compute entropy
        H = self._entropy(y)

        # reshape feature vector to shape (n,1)
        Xj = Xj.reshape((n,1))
        values = np.unique(Xj) # unique values in Xj, sorted
        n_values = len(values)

        # compute optimal conditional entropy by trying all thresholds
        thresholds = np.empty(n_values - 1) # possible thresholds
        H_conds = np.empty(n_values - 1)    # associated conditional entropies
        for i in xrange(n_values - 1) :
            threshold = (values[i] + values[i+1]) / 2.
            thresholds[i] = threshold

            X1, y1, X2, y2 = self._split_data(Xj, y, 0, threshold)

            H_cond1 = self._entropy(y1)
            H_cond2 = self._entropy(y2)
            H_conds[i] = float(len(X1))/float(len(Xj)) * H_cond1 + float(len(X2))/fl
oat(len(Xj)) * H_cond2

        # find minimium conditional entropy (maximum information gain)
        # and associated threshold
        best_H_cond = H_conds.min()
        indices = np.where(H_conds == best_H_cond)[0]
        best_index = np.random.choice(indices)
        best_threshold = thresholds[best_index]


        # compute information gain
        info_gain = H - best_H_cond

        return info_gain, best_threshold

    def _split_data(self, X, y, feature, threshold) :
        """
        Split dataset (X,y) into two datasets (X1,y1) and (X2,y2)
        based on feature and threshold.

        (X1,y1) contains the subset of (X,y) such that X[i,feature] <= threshold.
        (X2,y2) contains the subset of (X,y) such that X[i,feature] > threshold.

        Parameters
        --------------------
            X         -- numpy array of shape (n,d), samples
            y         -- numpy array of shape (n,), target classes
            feature   -- int, feature index to split on
            threshold -- float, feature threshold

        Returns
        --------------------
            X1        -- numpy array of shape (n1,d), samples
            y1        -- numpy array of shape (n1,), target classes
            X2        -- numpy array of shape (n2,d), samples
```

```
            y2         -- numpy array of shape (n2,), target classes
        """
        n, d = X.shape
        if n != len(y) :
            raise Exception("feature vector and label vector must have same length")

        X1, X2 = [], []
        y1, y2 = [], []
        ### ========== TODO : START ========== ###
        for i in range(len(X)):
            if X[i,feature] <= threshold:
                X1.append(X[i])
                y1.append(y[i])
            else:
                X2.append(X[i])
                y2.append(y[i])

        ### ========== TODO : END ========== ###
        X1, X2 = np.array(X1), np.array(X2)
        y1, y2 = np.array(y1), np.array(y2)

        return X1, y1, X2, y2

    def _choose_feature(self, X, y) :
        """
        Choose a feature with max information gain from (X,y).

        Parameters
        --------------------
            X             -- numpy array of shape (n,d), samples
            y             -- numpy array of shape (n,), target classes

        Returns
        --------------------
            best_feature   -- int, feature to split on
            best_threshold -- float, feature threshold
        """
        n, d = X.shape
        if n != len(y) :
            raise Exception("feature vector and label vector must have same length")

        # compute optimal information gain by trying all features
        thresholds = np.empty(d) # best threshold for each feature
        scores     = np.empty(d) # best information gain for each feature
        for j in xrange(d) :
            if (X[:,j] == X[0,j]).all() :
                # skip if all feature values equal
                score, threshold = -1, None # use an invalid (but numeric) score
            else :
                score, threshold = self._information_gain(X[:,j], y)
            thresholds[j] = threshold
            scores[j] = score

        # find maximum information gain
        # and associated feature and threshold
        best_score = scores.max()
        indices = np.where(scores == best_score)[0]
        best_feature = np.random.choice(indices)
        best_threshold = thresholds[best_feature]

        return best_feature, best_threshold

    def _create_new_node(self, node, feature, threshold, value, impurity) :
        """
        Create a new internal node.

        Parameters
        --------------------
            node      -- int, current node index
            feature   -- int, feature index to split on
            threshold -- float, feature threshold
            value     -- numpy array of shape (n_classes,), class counts of current
node
```

```
            impurity  -- float, impurity of current node
        """
        self.children_left[node]  = self._next_node
        self._next_node += 1
        self.children_right[node] = self._next_node
        self._next_node += 1

        self.feature[node]        = feature
        self.threshold[node]      = threshold
        self.value[node]          = value
        self.impurity[node]       = impurity
        self.n_node_samples[node] = sum(value)

    def _create_new_leaf(self, node, value, impurity) :
        """
        Create a new leaf node.

        Parameters
        --------------------
            node     -- int, current node index
            value    -- numpy array of shape (n_classes,), class counts of current n
ode
            impurity -- float, impurity of current node
        """
        self.children_left[node]  = Tree.TREE_LEAF
        self.children_right[node] = Tree.TREE_LEAF

        self.feature[node]        = Tree.TREE_UNDEFINED
        self.threshold[node]      = Tree.TREE_UNDEFINED
        self.value[node]          = value
        self.impurity[node]       = impurity
        self.n_node_samples[node] = sum(value)

    def _build_helper(self, X, y, node=0) :

        """
        Build a decision tree from (X,y) in depth-first fashion.

        Parameters
        --------------------
            X        -- numpy array of shape (n,d), samples
            y        -- numpy array of shape (n,), target classes
            node     -- int, current node index (index of root for current subtree)
        """

        n, d = X.shape
        value = self._get_value(y)
        impurity = self._entropy(y)

        ### ========== TODO : START ========== ###
        # part d: decision tree induction algorithm
        # you can modify any code within this TODO block

        # base case
        # 1) all samples have same labels
        equal_Y = True
        for label in range(len(y) - 1):
            if y[label] != y[label+1]:
                equal_Y = False
        equal_X = True
        for i in range(d):
            for j in range(n - 1):
                if X[j][i] != X[j+1][i]:
                    equal_X = False
        print "X:   " + str(X) + '\n'
        print "Is this all the same? :      " + str(equal_X)
        if equal_Y: #all function isn't working properly
            self._create_new_leaf(node, value, impurity)

        # 2) all feature values are equal
        elif equal_X: #all function isn't working properly
            self._create_new_leaf(node, value, impurity)
```

```
        else:


            # choose best feature (and find associated threshold)
            feature, threshold = self._choose_feature(X,y)
            # make new decision tree node
            self._create_new_node(node, feature, threshold, value, impurity)
            # split data on best feature
            X1, y1, X2, y2 = self._split_data(X,y, feature, threshold)
            # build left subtree using recursion
            self._build_helper(X1, y1, self.children_left[node])
            #self._build_helper(X1,y1, node + 1)
            # build right subtree using recursion
           # self._build_helper(X2,y2,node + 2)
            self._build_helper(X2, y2, self.children_right[node])


        ### ========== TODO : END ========== ###

    #=========================================
    # main functions

    def fit(self, X, y) :
        """
        Build a decision tree from (X,y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes

        Returns
        --------------------
            self -- an instance of self
        """

        # y must contain only integers
        if not np.equal(np.mod(y, 1), 0).all() :
            raise NotImplementedError("y must contain only integers")

        # store classes
        self._classes = np.unique(y)

        # build tree
        self._build_helper(X, y)

        # resize arrays
        self.node_count     = self._next_node
        self.children_left  = self.children_left[:self.node_count]
        self.children_right = self.children_right[:self.node_count]
        self.feature        = self.feature[:self.node_count]
        self.threshold      = self.threshold[:self.node_count]
        self.value          = self.value[:self.node_count]
        self.impurity       = self.impurity[:self.node_count]
        self.n_node_samples = self.n_node_samples[:self.node_count]

        return self

    def predict(self, X) :
        """
        Predict target for X.

        Parameters
        --------------------
            X -- numpy array of shape (n,d), samples

        Returns
        --------------------
            y -- numpy array of shape (n,n_classes), values
        """

        n, d = X.shape
```

```
        y = np.empty((n, self.n_classes))

        ### ========== TODO : START ========== ###
        # part e: make predictions

        # for each sample
        #   start at root of tree
        j = 0
        for sample in X:
            i = 0
            while self.children_left[i] != Tree.TREE_LEAF and self.children_right[i]
 != Tree.TREE_LEAF:
                feature = self.feature[i]
                if sample[feature] > self.threshold[i]:
                    i = self.children_right[i]
                else:
                    i = self.children_left[i]
            y[j] = self.value[i]
            j += 1


        #   follow edges to leaf node
        #   find value at leaf node

        ### ========== TODO : END ========== ###

        return y


class Classifier(object) :
    """
    Classifier interface.
    """

    def fit(self, X, y):
        raise NotImplementedError()

    def predict(self, X):
        raise NotImplementedError()


class DecisionTreeClassifier(Classifier) :

    def __init__(self, criterion="entropy", random_state=None) :
        """
        A decision tree classifier.

        Attributes
        --------------------
            classes_    -- numpy array of shape (n_classes, ), the classes labels
            n_classes_  -- int, the number of classes
            n_features_ -- int, the number of features
            n_outputs_  -- int, the number of outputs
            tree_       -- the underlying Tree object
        """
        if criterion != "entropy":
            raise NotImplementedError()

        self.n_features_ = None
        self.classes_    = None
        self.n_classes_  = None
        self.n_outputs_  = None
        self.tree_       = None
        self.random_state = random_state

    def fit(self, X, y) :
        """
        Build a decision tree classifier from the training set (X, y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes
```

```
        Returns
        --------------------
            self -- an instance of self
        """

        n_samples, self.n_features_ = X.shape

        # determine number of outputs
        if y.ndim != 1 :
            raise NotImplementedError("each sample must have a single label")
        self.n_outputs_ = 1

        # determine classes
        classes = np.unique(y)
        self.classes_ = classes
        self.n_classes_ = classes.shape[0]

        # set random state
        np.random.seed(self.random_state)

        # main
        self.tree_ = Tree(self.n_features_, self.n_classes_, self.n_outputs_)
        self.tree_.fit(X, y)
        return self

    def predict(self, X) :
        """
        Predict class value for X.

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples

        Returns
        --------------------
            y    -- numpy array of shape (n,), predicted classes
        """

        if self.tree_ is None :
            raise Exception("Classifier not initialized. Perform a fit first.")

        # defer to self.tree_
        X = X.astype(tree._tree.DTYPE)
        proba = self.tree_.predict(X)
        predictions = self.classes_.take(np.argmax(proba, axis=1), axis=0)
        return predictions


######################################################################
# functions
######################################################################

def load_movie_dataset():
    """Load movie dataset."""
    # Note: This is not a good representation (use one-hot encoding instead),
    #       but it is easier and sufficient for a toy dataset.
    # type:     animated = 0, comedy = 1, drama = 2
    # length:   short = 0, medium = 1, long = 2
    # director: adamson = 0, lasseter = 1, singer = 2
    # actors:   not famous = 0, famous = 1
    # liked:    no = 0, famous = 1
    data = np.array([[1, 0, 0, 0, 1],
                     [0, 0, 1, 0, 0],
                     [2, 1, 0, 0, 1],
                     [0, 2, 1, 1, 0],
                     [1, 2, 1, 1, 0],
                     [2, 1, 2, 1, 1],
                     [0, 0, 2, 0, 1],
                     [1, 2, 0, 1, 1],
                     [2, 1, 1, 0, 1]])
    names = ['type', 'length', 'director', 'famous_actor', 'liked']
```

```
    X = data[:,:-1]
    Xnames = names[:-1]
    y = data[:,-1]
    yname = names[-1]

    return X, y, Xnames, yname


def print_tree(decision_tree, feature_names=None, class_names=None, root=0, depth=1)
:
    """
    Print decision tree.

    Only works with decision_tree.n_outputs = 1.
    https://healthyalgorithms.com/2015/02/19/ml-in-python-getting-the-decision-tree-
out-of-sklearn/

    Parameters
    --------------------
        decision_tree -- tree (sklearn.tree._tree.Tree or Tree)
        feature_names -- list, feature names
        class_names   -- list, class names
    """

    t = decision_tree
    if t.n_outputs != 1:
        raise NotImplementedError()

    if depth == 1:
        print 'def predict(x):'

    indent = '    ' * depth

    # determine node numbers of children
    left_child = t.children_left[root]
    right_child = t.children_right[root]

    # determine predicted class for this node
    values = t.value[root][0]
    class_ndx = np.argmax(values)
    if class_names is not None:
        class_str = class_names[class_ndx]
    else:
        class_str = str(class_ndx)

    # determine node string
    node_str = "(node %d: impurity = %.2f, samples = %d, value = %s, class = %s)" %
\
         (root, t.impurity[root], t.n_node_samples[root], values, class_str)

    # main code
    if left_child == tree._tree.TREE_LEAF:
        print indent + 'return %s # %s' % (class_str, node_str)
    else:
        # determine feature name
        if feature_names is not None:
            name = feature_names[t.feature[root]]
        else:
            name = "x_%d" % t.feature[root]

        print indent + 'if %s <= %.2f: # %s' % (name, t.threshold[root], node_str)
        print_tree(t, feature_names, class_names, root=left_child, depth=depth+1)

        print indent + 'else:'
        print_tree(t, feature_names, class_names, root=right_child, depth=depth+1)


######################################################################
# main
######################################################################

def main():
    np.random.seed(1234)
```

```
    # load movie dataset
    X, y, Xnames, yname = load_movie_dataset()

    #========================================
    # scikit-learn DecisionTreeClassifier
    print 'Using DecisionTreeClassifier from scikit-learn...'

    from sklearn.tree import DecisionTreeClassifier as DTC
    clf = DTC(criterion='entropy', random_state=1234)
    clf.fit(X, y)
    print_tree(clf.tree_, feature_names=Xnames, class_names=["No", "Yes"])
    y_pred = clf.predict(X)
    print 'y_pred = ', y_pred

    """
    Output

    def predict(x):
        if director <= 0.50: # (node 0: impurity = 0.92, samples = 9, value = [ 3.
6.], class = Yes)
            return Yes # (node 1: impurity = 0.00, samples = 3, value = [ 0.  3.], c
lass = Yes)
        else:
            if type <= 1.50: # (node 2: impurity = 1.00, samples = 6, value = [ 3.
3.], class = No)
                if director <= 1.50: # (node 3: impurity = 0.81, samples = 4, value
= [ 3.  1.], class = No)
                    return No # (node 4: impurity = 0.00, samples = 3, value = [ 3.
 0.], class = No)
                else:
                    return Yes # (node 5: impurity = 0.00, samples = 1, value = [ 0.
  1.], class = Yes)
            else:
                return Yes # (node 6: impurity = 0.00, samples = 2, value = [ 0.  2.
], class = Yes)
    y_pred =  [1 0 1 0 0 1 1 1 1]
    """

    """
    # save the classifier -- requires GraphViz and pydot
    import StringIO, pydot
    dot_data = StringIO.StringIO()
    tree.export_graphviz(clf, out_file=dot_data,
                         feature_names=Xnames,
                         class_names=["No", "Yes"])
    graph = pydot.graph_from_dot_data(dot_data.getvalue())
    #graph.write_pdf("dtree_movie.pdf")
    """

    print

    #========================================
    # home-grown DecisionTreeClassifier
    print 'Using my DecisionTreeClassifier...'

    # test cases
    n_features = X.shape[1]
    n_classes = len(np.unique(y))
    my_tree = Tree(n_features, n_classes, 1)

    # _entropy -> entropy
    # soln -- 0.918295834054
    print 'H =', my_tree._entropy(y)

    # _split_data -> X1, y1, X2, y2
    # soln --
    #   [X1,y1] = [[1 0 0 0 1]    [X2,y2] = [[2 1 0 0 1]
    #              [0 0 1 0 0]               [2 1 2 1 1]
    #              [0 2 1 1 0]               [2 1 1 0 1]]
    #              [1 2 1 1 0]
    #              [0 0 2 0 1]
    #              [1 2 0 1 1]]
```

```
    X1, y1, X2, y2 = my_tree._split_data(X, y, 0, 1.5)
    print '[X1,y1] =\n', np.column_stack((X1, y1))
    print '[X2,y2] =\n', np.column_stack((X2, y2))


    # _information_gain -> information gain, threshold
    # soln -- (0.25162916738782293, 1.5)
    print '(I,t) =', my_tree._information_gain(X[:,0], y)
    # main
    # soln -- See below. You may get a different decision tree but y_pred should be
the same.
    clf2 = DecisionTreeClassifier(random_state=1234)
    clf2.fit(X, y)
    print_tree(clf2.tree_, feature_names=Xnames, class_names=["No", "Yes"])

    y_pred2 = clf2.predict(X)
    print 'y_pred2 =', y_pred2

    assert (y_pred == y_pred2).all(), "predictions are not the same"

    """
     Output

    def predict(x):
        if director <= 0.50: # (node 0: impurity = 0.92, samples = 9, value = [ 3.
6.], class = Yes)
            return Yes # (node 1: impurity = 0.00, samples = 3, value = [ 0.  3.], c
lass = Yes)
        else:
            if director <= 1.50: # (node 2: impurity = 1.00, samples = 6, value = [
3.  3.], class = No)
                if type <= 1.50: # (node 3: impurity = 0.81, samples = 4, value = [
3.  1.], class = No)
                    return No # (node 5: impurity = 0.00, samples = 3, value = [ 3.
 0.], class = No)
                else:
                    return Yes # (node 6: impurity = 0.00, samples = 1, value = [ 0.
  1.], class = Yes)
            else:
                return Yes # (node 4: impurity = 0.00, samples = 2, value = [ 0.  2.
], class = Yes)
    y_pred2 = [1 0 1 0 0 1 1 1 1]
    """

    print

    #========================================
    # train Decision Tree classifier on Titanic data
    print 'Classifying Titanic data set...'

    titanic = load_data("titanic_train.csv", header=1, predict_col=0)
    X = titanic.X
    y = titanic.y

    clf = DTC(criterion='entropy')
    clf.fit(X, y)
    y_pred = clf.predict(X)

    clf2 = DecisionTreeClassifier()
    clf2.fit(X, y)
    y_pred2 = clf2.predict(X)

    print y_pred
    print y_pred2
    assert (y_pred == y_pred2).all(), "predictions are not the same"

    #========================================

    print 'Done'

if __name__ == "__main__":
    main()
```