01/30/18 10:10:27 /Users/caiglencross/Documents/MachineLearning/ps2/source/dtree.py

```
  1 """
  2 Author      : Cai Glencross and Katie Li
  3 Class       : HMC CS 158
  4 Date        : 2018 Jan 30
  5 Description : Decision Tree Classifier
  6 """
  7
  8 # Use only the provided packages!
  9 import collections
 10 from util import *
 11
 12 # numpy libraries
 13 import numpy as np
 14
 15 # scikit-learn libraries
 16 from sklearn import tree
 17
 18 ##############################################################################
 19 # classes
 20 ##############################################################################
 21
 22 class Tree(object) :
 23     """
 24     Array-based representation of a binary decision tree.
 25
 26     See tree._tree.Tree (a Python wrapper around a C class).
 27     The binary tree is represented as a number of parallel arrays. The i-th
 28     element of each array holds information about the node 'i'. Node 0 is the
 29     tree's root. NOTE: Some of the arrays only apply to either leaves or split
 30     nodes, resp. In this case the values of nodes of the other type are
 31     arbitrary!
 32
 33     Attributes
 34     ----------
 35       node_count : int
 36           The number of nodes (internal nodes + leaves) in the tree.
 37
 38       children_left : array of int, shape [node_count]
 39           children_left[i] holds the node id of the left child of node i.
 40           For leaves, children_left[i] == TREE_LEAF. Otherwise,
 41           children_left[i] > i. This child handles the case where
 42           X[:, feature[i]] <= threshold[i].
 43
 44       children_right : array of int, shape [node_count]
 45           children_right[i] holds the node id of the right child of node i.
 46           For leaves, children_right[i] == TREE_LEAF. Otherwise,
 47           children_right[i] > i. This child handles the case where
 48           X[:, feature[i]] > threshold[i].
 49
 50       feature : array of int, shape [node_count]
 51           feature[i] holds the feature to split on, for the internal node i.
 52
 53       threshold : array of double, shape [node_count]
 54           threshold[i] holds the threshold for the internal node i.
 55
 56       value : array of double, shape [node_count, 1, max_n_classes]
 57           value[i][0] holds the counts of each class reaching node i
```

```
58
59          impurity : array of double, shape [node_count]
60              impurity[i] holds the impurity at node i.
61
62          n_node_samples : array of int, shape [node_count]
63              n_node_samples[i] holds the number of training samples reaching node
    i.
64      """
65      TREE_LEAF = tree._tree.TREE_LEAF
66      TREE_UNDEFINED = tree._tree.TREE_UNDEFINED
67
68      def __init__(self, n_features, n_classes, n_outputs=1) :
69          if n_outputs != 1 :
70              raise NotImplementedError("each sample must have a single label")
71
72          self.n_features     = n_features
73          self.n_classes      = n_classes
74          self.n_outputs      = n_outputs
75
76          capacity = 2047 # arbitrary, allows max_depth = 10
77          self.node_count     = capacity
78          self.children_left  = np.empty(self.node_count, dtype=int)
79          self.children_right = np.empty(self.node_count, dtype=int)
80          self.feature        = np.empty(self.node_count, dtype=int)
81          self.threshold      = np.empty(self.node_count)
82          self.value          = np.empty((self.node_count, n_outputs, n_classes))
83          self.impurity       = np.empty(self.node_count)
84          self.n_node_samples = np.empty(self.node_count, dtype=int)
85
86          # private
87          self._next_node     = 1 # start at root
88          self._classes       = None
89
90      #=========================================
91      # helper functions
92
93      def _get_value(self, y) :
94          """
95          Get count of each class.
96
97          Parameters
98          ----------
99              y     -- numpy array of shape (n,), target classes
100
101         Returns
102         -------
103             value -- numpy array of shape (n_classes,), class counts
104                      value[i] holds count of each class
105         """
106         if len(y) == 0 :
107             raise Exception("cannot separate empty set")
108
109         counter = collections.defaultdict(lambda: 0)
110         for label in y :
111             counter[label] += 1
112
113         value = np.empty((self.n_classes,))
114         for i, label in enumerate(self._classes) :
115             value[i] = counter[label]
116
```

```
117         return value
118 
119     def _entropy(self, y) :
120         """
121         Compute entropy.
122 
123         Parameters
124         --------------------
125             y -- numpy array of shape (n,), target classes
126 
127         Returns
128         --------------------
129             H -- entropy
130         """
131 
132         # compute counts
133         _, counts = np.unique(y, return_counts=True)
134 
135         ### ========== TODO : START ========== ###
136         # part a: compute entropy
137         # hint: use np.log2 to take log
138         H = 0
139         total_sum = np.sum(counts)
140         for c in counts:
141             prob_y = float(c)/total_sum
142             H += -1.0* prob_y * np.log2(prob_y)
143         ### ========== TODO : END ========== ###
144 
145         return H
146 
147     def _information_gain(self, Xj, y) :
148         """
149         Compute information gain.
150 
151         Parameters
152         --------------------
153             Xj             -- numpy array of shape (n,), samples (one feature only)
154             y              -- numpy array of shape (n,), target classes
155 
156         Returns
157         --------------------
158             info_gain      -- float, information gain using best threshold
159             best_threshold -- float, threshold with best information gain
160         """
161         n = len(Xj)
162         if n != len(y) :
163             raise Exception("feature vector and class vector must have same length")
164 
165         # compute entropy
166         H = self._entropy(y)
167 
168         # reshape feature vector to shape (n,1)
169         Xj = Xj.reshape((n,1))
170         values = np.unique(Xj) # unique values in Xj, sorted
171         n_values = len(values)
172 
173         # compute optimal conditional entropy by trying all thresholds
174         thresholds = np.empty(n_values - 1) # possible thresholds
```

```
175         H_conds = np.empty(n_values - 1)    # associated conditional entropies
176         for i in xrange(n_values - 1) :
177             threshold = (values[i] + values[i+1]) / 2.
178             thresholds[i] = threshold
179 
180             X1, y1, X2, y2 = self._split_data(Xj, y, 0, threshold)
181             ### ========== TODO : START ========== ###
182             # part c: compute conditional entropy
183             H_cond = 0
184             H_1 = self._entropy(y1)
185             H_2 = self._entropy(y2)
186             H_cond = (float(len(X1))/n)*H_1 + (float(len(X2))/n)*H_2
187 
188             ### ========== TODO : END ========== ###
189             H_conds[i] = H_cond
190 
191         # find minimium conditional entropy (maximum information gain)
192         # and associated threshold
193         best_H_cond = H_conds.min()
194         indices = np.where(H_conds == best_H_cond)[0]
195         best_index = np.random.choice(indices)
196         best_threshold = thresholds[best_index]
197 
198         # compute information gain
199         info_gain = H - best_H_cond
200 
201         return info_gain, best_threshold
202 
203     def _split_data(self, X, y, feature, threshold) :
204         """
205         Split dataset (X,y) into two datasets (X1,y1) and (X2,y2)
206         based on feature and threshold.
207 
208         (X1,y1) contains the subset of (X,y) such that X[i,feature] <=
    threshold.
209         (X2,y2) contains the subset of (X,y) such that X[i,feature] > threshold.
210 
211         Parameters
212         --------------------
213             X         -- numpy array of shape (n,d), samples
214             y         -- numpy array of shape (n,), target classes
215             feature   -- int, feature index to split on
216             threshold -- float, feature threshold
217 
218         Returns
219         --------------------
220             X1        -- numpy array of shape (n1,d), samples
221             y1        -- numpy array of shape (n1,), target classes
222             X2        -- numpy array of shape (n2,d), samples
223             y2        -- numpy array of shape (n2,), target classes
224         """
225         n, d = X.shape
226         if n != len(y) :
227             raise Exception("feature vector and label vector must have same
    length")
228 
229         X1, X2 = [], []
230         y1, y2 = [], []
231         ### ========== TODO : START ========== ###
232 
```

```
233         #TODO: filter on the matching indicies where feature level is gt or lt
234         X1 = X[X[:,feature] <= threshold,:]
235         y1 = y[X[:,feature] <= threshold]
236 
237         X2 = X[X[:,feature] > threshold, :]
238         y2 = y[X[:,feature] > threshold]
239 
240         ### ========== TODO : END ========== ###
241         X1, X2 = np.array(X1), np.array(X2)
242         y1, y2 = np.array(y1), np.array(y2)
243 
244         return X1, y1, X2, y2
245 
246     def _choose_feature(self, X, y) :
247         """
248         Choose a feature with max information gain from (X,y).
249 
250         Parameters
251         --------------------
252             X             -- numpy array of shape (n,d), samples
253             y             -- numpy array of shape (n,), target classes
254 
255         Returns
256         --------------------
257             best_feature   -- int, feature to split on
258             best_threshold -- float, feature threshold
259         """
260         n, d = X.shape
261         if n != len(y) :
262             raise Exception("feature vector and label vector must have same
    length")
263 
264         # compute optimal information gain by trying all features
265         thresholds = np.empty(d) # best threshold for each feature
266         scores     = np.empty(d) # best information gain for each feature
267         for j in xrange(d) :
268             if (X[:,j] == X[0,j]).all() :
269                 # skip if all feature values equal
270                 score, threshold = -1, None # use an invalid (but numeric) score
271             else :
272                 score, threshold = self._information_gain(X[:,j], y)
273             thresholds[j] = threshold
274             scores[j] = score
275 
276         # find maximum information gain
277         # and associated feature and threshold
278         best_score = scores.max()
279         indices = np.where(scores == best_score)[0]
280         best_feature = np.random.choice(indices)
281         best_threshold = thresholds[best_feature]
282 
283         return best_feature, best_threshold
284 
285     def _create_new_node(self, node, feature, threshold, value, impurity) :
286         """
287         Create a new internal node.
288 
289         Parameters
290         --------------------
291             node      -- int, current node index
```

```
292             feature   -- int, feature index to split on
293             threshold -- float, feature threshold
294             value     -- numpy array of shape (n_classes,), class counts of
    current node
295             impurity  -- float, impurity of current node
296         """
297         self.children_left[node]  = self._next_node
298         self._next_node += 1
299         self.children_right[node] = self._next_node
300         self._next_node += 1
301 
302         self.feature[node]        = feature
303         self.threshold[node]      = threshold
304         self.value[node]          = value
305         self.impurity[node]       = impurity
306         self.n_node_samples[node] = sum(value)
307 
308     def _create_new_leaf(self, node, value, impurity) :
309         """
310         Create a new leaf node.
311 
312         Parameters
313         ----------
314             node     -- int, current node index
315             value    -- numpy array of shape (n_classes,), class counts of
    current node
316             impurity -- float, impurity of current node
317         """
318         self.children_left[node]  = Tree.TREE_LEAF
319         self.children_right[node] = Tree.TREE_LEAF
320 
321         self.feature[node]        = Tree.TREE_UNDEFINED
322         self.threshold[node]      = Tree.TREE_UNDEFINED
323         self.value[node]          = value
324         self.impurity[node]       = impurity
325         self.n_node_samples[node] = sum(value)
326 
327     def _build_helper(self, X, y, node=0) :
328         """
329         Build a decision tree from (X,y) in depth-first fashion.
330 
331         Parameters
332         ----------
333             X        -- numpy array of shape (n,d), samples
334             y        -- numpy array of shape (n,), target classes
335             node     -- int, current node index (index of root for current
    subtree)
336         """
337 
338         n, d = X.shape
339 
340         value = self._get_value(y)
341         impurity = self._entropy(y)
342 
343         ### ========== TODO : START ========== ###
344         # part d: decision tree induction algorithm
345         # you can modify any code within this TODO block
346 
347         # base case
348         # 1) all samples have same labels
```

```
349         # 2) all feature values are equal
350         if impurity == 0 or len(np.unique(X, axis=0))==1: # you should modify
    this condition
351             # this line is so that the code can run
352             # you can comment it out (or not) once you add your own code
353             #pass
354 
355             # create a single leaf
356 
357             self._create_new_leaf(node, value, impurity)
358         else:
359             # this line is so that the code can run
360             # you can comment it out (or not) once you add your own code
361             # pass
362 
363             # choose best feature (and find associated threshold)
364             best_feature, best_threshold = self._choose_feature(X, y)
365             # make new decision tree node
366             self._create_new_node(node, best_feature, best_threshold, value,
    impurity)
367             # split data on best feature
368             X1, y1, X2, y2 = self._split_data(X,y,best_feature, best_threshold)
369             # build left subtree using recursion
370             self._build_helper(X1, y1, self.children_left[node])
371             # build right subtree using recursion
372             self._build_helper(X2, y2, self.children_right[node])
373         ### ========== TODO : END ========== ###
374 
375     #========================================
376     # main functions
377 
378     def fit(self, X, y) :
379         """
380         Build a decision tree from (X,y).
381 
382         Parameters
383         ----------
384             X    -- numpy array of shape (n,d), samples
385             y    -- numpy array of shape (n,), target classes
386 
387         Returns
388         ----------
389             self -- an instance of self
390         """
391 
392         # y must contain only integers
393         if not np.equal(np.mod(y, 1), 0).all() :
394             raise NotImplementedError("y must contain only integers")
395 
396         # store classes
397         self._classes = np.unique(y)
398 
399         # build tree
400         self._build_helper(X, y)
401 
402         # resize arrays
403         self.node_count     = self._next_node
404         self.children_left  = self.children_left[:self.node_count]
405         self.children_right = self.children_right[:self.node_count]
406         self.feature        = self.feature[:self.node_count]
```

```
407             self.threshold      = self.threshold[:self.node_count]
408             self.value          = self.value[:self.node_count]
409             self.impurity       = self.impurity[:self.node_count]
410             self.n_node_samples = self.n_node_samples[:self.node_count]
411
412             return self
413
414         def predict(self, X) :
415             """
416             Predict target for X.
417
418             Parameters
419             ----------
420                 X -- numpy array of shape (n,d), samples
421
422             Returns
423             -------
424                 y -- numpy array of shape (n,n_classes), values
425             """
426
427             n, d = X.shape
428             y = np.empty((n, self.n_classes))
429
430             ### ========== TODO : START ========== ###
431             # part e: make predictions
432
433             # for each sample
434             #   start at root of tree
435             #   follow edges to leaf node
436             #   find value at leaf node
437             #print "feature vector: ", self.feature
438             #print "value vector: ", self.value
439             for i in range(0,n):
440                 sample = X[i,:]
441                 #print sample
442                 current_node = 0
443                 while(self.children_left[current_node] != -1):
444                     current_feature = self.feature[current_node]
445                     current_threshold = self.threshold[current_node]
446                     #print "current node: ", current_node
447                     #print "current feature: ", current_feature
448                     #print "current threshold: ", current_threshold
449                     if sample[current_feature]<= current_threshold:
450                         current_node = self.children_left[current_node]
451                         #print "splitting left!"
452                     else:
453                         current_node = self.children_right[current_node]
454                         #print "splitting right!"
455                 #print "current node at end: ", current_node
456                 #print self.value[current_node]
457                 y[i] = self.value[current_node]
458
459
460             ### ========== TODO : END ========== ###
461
462             return y
463
464
465     class Classifier(object) :
466         """
```

```
467     Classifier interface.
468     """
469
470     def fit(self, X, y):
471         raise NotImplementedError()
472
473     def predict(self, X):
474         raise NotImplementedError()
475
476
477 class DecisionTreeClassifier(Classifier) :
478
479     def __init__(self, criterion="entropy", random_state=None) :
480         """
481         A decision tree classifier.
482
483         Attributes
484         ----------
485             classes_    -- numpy array of shape (n_classes, ), the classes
    labels
486             n_classes_  -- int, the number of classes
487             n_features_ -- int, the number of features
488             n_outputs_  -- int, the number of outputs
489             tree_       -- the underlying Tree object
490         """
491         if criterion != "entropy":
492             raise NotImplementedError()
493
494         self.n_features_ = None
495         self.classes_    = None
496         self.n_classes_  = None
497         self.n_outputs_  = None
498         self.tree_       = None
499         self.random_state = random_state
500
501     def fit(self, X, y) :
502         """
503         Build a decision tree classifier from the training set (X, y).
504
505         Parameters
506         ----------
507             X    -- numpy array of shape (n,d), samples
508             y    -- numpy array of shape (n,), target classes
509
510         Returns
511         ----------
512             self -- an instance of self
513         """
514
515         n_samples, self.n_features_ = X.shape
516
517         # determine number of outputs
518         if y.ndim != 1 :
519             raise NotImplementedError("each sample must have a single label")
520         self.n_outputs_ = 1
521
522         # determine classes
523         classes = np.unique(y)
524         self.classes_ = classes
525         self.n_classes_ = classes.shape[0]
```

```
526
527         # set random state
528         np.random.seed(self.random_state)
529
530         # main
531         self.tree_ = Tree(self.n_features_, self.n_classes_, self.n_outputs_)
532         self.tree_.fit(X, y)
533         return self
534
535     def predict(self, X) :
536         """
537         Predict class value for X.
538
539         Parameters
540         ----------
541             X    -- numpy array of shape (n,d), samples
542
543         Returns
544         -------
545             y    -- numpy array of shape (n,), predicted classes
546         """
547
548         if self.tree_ is None :
549             raise Exception("Classifier not initialized. Perform a fit first.")
550
551         # defer to self.tree_
552         X = X.astype(tree._tree.DTYPE)
553         proba = self.tree_.predict(X)
554         predictions = self.classes_.take(np.argmax(proba, axis=1), axis=0)
555         return predictions
556
557
558 ##########################################################################
559 # functions
560 ##########################################################################
561
562 def load_movie_dataset():
563     """Load movie dataset."""
564     # Note: This is not a good representation (use one-hot encoding instead),
565     #       but it is easier and sufficient for a toy dataset.
566     # type:     animated = 0, comedy = 1, drama = 2
567     # length:   short = 0, medium = 1, long = 2
568     # director: adamson = 0, lasseter = 1, singer = 2
569     # actors:   not famous = 0, famous = 1
570     # liked:    no = 0, famous = 1
571     data = np.array([[1, 0, 0, 0, 1],
572                      [0, 0, 1, 0, 0],
573                      [2, 1, 0, 0, 1],
574                      [0, 2, 1, 1, 0],
575                      [1, 2, 1, 1, 0],
576                      [2, 1, 2, 1, 1],
577                      [0, 0, 2, 0, 1],
578                      [1, 2, 0, 1, 1],
579                      [2, 1, 1, 0, 1]])
580     names = ['type', 'length', 'director', 'famous_actor', 'liked']
581
582     X = data[:,:-1]
583     Xnames = names[:-1]
584     y = data[:,-1]
585     yname = names[-1]
```

```
586
587     return X, y, Xnames, yname
588
589
590 def print_tree(decision_tree, feature_names=None, class_names=None, root=0,
    depth=1):
591     """
592     Print decision tree.
593
594     Only works with decision_tree.n_outputs = 1.
595     https://healthyalgorithms.com/2015/02/19/ml-in-python-getting-the-decision-
    tree-out-of-sklearn/
596
597     Parameters
598     ----------
599         decision_tree -- tree (sklearn.tree._tree.Tree or Tree)
600         feature_names -- list, feature names
601         class_names   -- list, class names
602     """
603
604     t = decision_tree
605     if t.n_outputs != 1:
606         raise NotImplementedError()
607
608     if depth == 1:
609         print 'def predict(x):'
610
611     indent = '    ' * depth
612
613     # determine node numbers of children
614     left_child = t.children_left[root]
615     right_child = t.children_right[root]
616
617     # determine predicted class for this node
618     values = t.value[root][0]
619     class_ndx = np.argmax(values)
620     if class_names is not None:
621         class_str = class_names[class_ndx]
622     else:
623         class_str = str(class_ndx)
624
625     # determine node string
626     node_str = "(node %d: impurity = %.2f, samples = %d, value = %s, class =
    %s)" % \
627         (root, t.impurity[root], t.n_node_samples[root], values, class_str)
628
629     # main code
630     if left_child == tree._tree.TREE_LEAF:
631         print indent + 'return %s # %s' % (class_str, node_str)
632     else:
633         # determine feature name
634         if feature_names is not None:
635             name = feature_names[t.feature[root]]
636         else:
637             name = "x_%d" % t.feature[root]
638
639         print indent + 'if %s <= %.2f: # %s' % (name, t.threshold[root],
    node_str)
640         print_tree(t, feature_names, class_names, root=left_child,
    depth=depth+1)
```

```
641
642         print indent + 'else:'
643         print_tree(t, feature_names, class_names, root=right_child,
    depth=depth+1)
644
645
646 ######################################################################
647 # main
648 ######################################################################
649
650 def main():
651     np.random.seed(1234)
652
653     # load movie dataset
654     X, y, Xnames, yname = load_movie_dataset()
655
656     #========================================
657     # scikit-learn DecisionTreeClassifier
658     print 'Using DecisionTreeClassifier from scikit-learn...'
659
660     from sklearn.tree import DecisionTreeClassifier as DTC
661     clf = DTC(criterion='entropy', random_state=1234)
662     clf.fit(X, y)
663     print_tree(clf.tree_, feature_names=Xnames, class_names=["No", "Yes"])
664     y_pred = clf.predict(X)
665     print 'y_pred = ', y_pred
666
667     """
668     Output
669
670     def predict(x):
671         if director <= 0.50: # (node 0: impurity = 0.92, samples = 9, value = [
    3.  6.], class = Yes)
672             return Yes # (node 1: impurity = 0.00, samples = 3, value = [ 0.
    3.], class = Yes)
673         else:
674             if type <= 1.50: # (node 2: impurity = 1.00, samples = 6, value = [
    3.  3.], class = No)
675                 if director <= 1.50: # (node 3: impurity = 0.81, samples = 4,
    value = [ 3.  1.], class = No)
676                     return No # (node 4: impurity = 0.00, samples = 3, value = [
    3.  0.], class = No)
677                 else:
678                     return Yes # (node 5: impurity = 0.00, samples = 1, value =
    [ 0.  1.], class = Yes)
679             else:
680                 return Yes # (node 6: impurity = 0.00, samples = 2, value = [
    0.  2.], class = Yes)
681     y_pred =  [1 0 1 0 0 1 1 1 1]
682     """
683
684     """
685     # save the classifier -- requires GraphViz and pydot
686     import StringIO, pydot
687     dot_data = StringIO.StringIO()
688     tree.export_graphviz(clf, out_file=dot_data,
689                          feature_names=Xnames,
690                          class_names=["No", "Yes"])
691     graph = pydot.graph_from_dot_data(dot_data.getvalue())
692     #graph.write_pdf("dtree_movie.pdf")
```

```
693     """
694
695     print
696
697     #========================================
698     # home-grown DecisionTreeClassifier
699     print 'Using my DecisionTreeClassifier...'
700
701     # test cases
702     n_features = X.shape[1]
703     n_classes = len(np.unique(y))
704     my_tree = Tree(n_features, n_classes, 1)
705
706     # _entropy -> entropy
707     # soln -- 0.918295834054
708     print 'H =', my_tree._entropy(y)
709
710     # _split_data -> X1, y1, X2, y2
711     # soln --
712     #   [X1,y1] = [[1 0 0 0 1]    [X2,y2] = [[2 1 0 0 1]
713     #              [0 0 1 0 0]               [2 1 2 1 1]
714     #              [0 2 1 1 0]               [2 1 1 0 1]]
715     #              [1 2 1 1 0]
716     #              [0 0 2 0 1]
717     #              [1 2 0 1 1]]
718     X1, y1, X2, y2 = my_tree._split_data(X, y, 0, 1.5)
719     print '[X1,y1] =\n', np.column_stack((X1, y1))
720     print '[X2,y2] =\n', np.column_stack((X2, y2))
721
722     # _information_gain -> information gain, threshold
723     # soln -- (0.25162916738782293, 1.5)
724     print '(I,t) =', my_tree._information_gain(X[:,0], y)
725
726     # main
727     # soln -- See below. You may get a different decision tree but y_pred should
    be the same.
728     clf2 = DecisionTreeClassifier(random_state=1234)
729     clf2.fit(X, y)
730     print_tree(clf2.tree_, feature_names=Xnames, class_names=["No", "Yes"])
731     y_pred2 = clf2.predict(X)
732     print 'y_pred2 =', y_pred2
733
734     assert (y_pred == y_pred2).all(), "predictions are not the same"
735
736     """
737     Output
738
739     def predict(x):
740         if director <= 0.50: # (node 0: impurity = 0.92, samples = 9, value = [
    3.  6.], class = Yes)
741             return Yes # (node 1: impurity = 0.00, samples = 3, value = [ 0.
    3.], class = Yes)
742         else:
743             if director <= 1.50: # (node 2: impurity = 1.00, samples = 6, value
    = [ 3.  3.], class = No)
744                 if type <= 1.50: # (node 3: impurity = 0.81, samples = 4, value
    = [ 3.  1.], class = No)
745                     return No # (node 5: impurity = 0.00, samples = 3, value = [
    3.  0.], class = No)
746                 else:
```

```
747                     return Yes # (node 6: impurity = 0.00, samples = 1, value =
    [ 0.  1.], class = Yes)
748             else:
749                 return Yes # (node 4: impurity = 0.00, samples = 2, value = [
    0.  2.], class = Yes)
750     y_pred2 = [1 0 1 0 0 1 1 1 1]
751     """
752
753     print
754
755     #========================================
756     # train Decision Tree classifier on Titanic data
757     print 'Classifying Titanic data set...'
758
759     titanic = load_data("titanic_train.csv", header=1, predict_col=0)
760     X = titanic.X
761     y = titanic.y
762
763     clf = DTC(criterion='entropy')
764     clf.fit(X, y)
765     y_pred = clf.predict(X)
766
767     clf2 = DecisionTreeClassifier()
768     clf2.fit(X, y)
769     y_pred2 = clf2.predict(X)
770
771     assert (y_pred == y_pred2).all(), "predictions are not the same"
772
773     #========================================
774
775     print 'Done'
776
777
778 if __name__ == "__main__":
779     main()
```