103590450 四資四 馬茂源

```
[1]   from sklearn.datasets import load_iris
      from sklearn.metrics import accuracy_score
      from sklearn.model_selection import train_test_split
      from sklearn.decomposition import FastICA
      from scipy.stats import multivariate_normal, norm
      import numpy as np
      import itertools
```

```
[2]   class MyKNeighborsClassifier:

          def __init__(self, n_neighbors=3, **kwargs):
              self._k = n_neighbors
              self._X = self._y = None
              self.set_params(**kwargs)

          def get_params(self, deep=True):
              # suppose this estimator has parameters "alpha" and "recursive"
              return self.__dict__

          def set_params(self, **parameters):
              for parameter, value in parameters.items():
                  setattr(self, parameter, value)
              return self

          def fit(self, X, y):
              self._X = X.copy()
              self._y = y.copy()

          def _predict(self, x):
              distances = np.apply_along_axis(lambda x1: np.linalg.norm(x-x1),
                                              1, self._X)
              X_candidates = np.argsort(distances)[:self._k]
              y_candidates = self._y[X_candidates]
              return np.argmax(np.bincount(y_candidates.astype('int64')))

          def score(self, X, y_true):
              return accuracy_score(y_true, self.predict(X))

          def predict(self, X):
              return np.apply_along_axis(lambda x: self._predict(x), 1, X)
```

```
[3]   iris = load_iris()
      feature_names = iris.feature_names.copy()
      iris_X = iris.data
```

```
iris_y = iris.target
print(iris_X.shape, iris_y.shape)
```

```
(150, 4) (150,)
```

1 . In this problem, you are asked to use ICA on Iris dataset for dimensionality reduction before classification. To simplify the problem, you do not need to implement the ICA program.

Instead, find an existing one and learn how to use it. As you may not be able to store internal parameters of the ICA, input all of the 150 samples to find the corresponding independent components as the preprocessing step.

You may assume that there are four sources and four observations. On the obtained four components, pick the two components with largest energy as new features. Randomly pick 70 % of the samples (represented by new features) as training set and the rest as test set. Implement the 3-NN classifier to compute the accuracy. Repeat the drawing and the 3-NN classification 10 times and compute the average accuracy and accuracy variance. For simplicity, use the Euclidean distance in the k-NN computation.

```
[4]  ica = FastICA(n_components=2)
     ica_X = ica.fit_transform(iris_X)
```

```
/usr/local/lib/python3.6/site-packages/scipy/linalg/basic.py:1226:
RuntimeWarning: internal gelsd driver lwork query error, required iwork
dimension not returned. This is likely the result of LAPACK bug 0038,
fixed in LAPACK 3.2.2 (released July 21, 2010). Falling back to 'gelss'
driver.
  warnings.warn(mesg, RuntimeWarning)
```

```
[5]  acc = []
     for i in range(10):
         train_X, test_X, train_y, test_y = train_test_split(ica_X,
                                                             iris_y,
                                                             train_size=0.7)
         model = MyKNeighborsClassifier()
         model.fit(train_X, train_y)
         acc.append(model.score(test_X, test_y))
     print('avg acc:{}, variance of acc:{}'.format(np.mean(acc), np.var(acc)))
```

```
/usr/local/lib/python3.6/site-
packages/sklearn/model_selection/_split.py:2026: FutureWarning: From
```

```
version 0.21, test_size will always complement train_size unless both are
specified.
  FutureWarning)
avg acc:0.9511111111111111, variance of acc:0.0010666666666666665
```

2 . We learned the k-means for clustering in the lecture. Implement the algorithm with the Iris dataset.

In this problem, we know k = 3. Use the first sample in each class as the initial cluster center to do the clustering.

Remember that the cluster centers are points in 4-dimensional space. To have a unique answer, use the same sequence given in the dataset to feed into your program.

That is, do not shuffler the dataset. Once your program converges,

[6]

```python
class MyKmeans:

    def __init__(self, n_clusters=3, init=None, max_iter=300, tol=1e-4):
        self.n_clusters = n_clusters
        self.centroids = init.copy()
        self.max_iter = max_iter
        self.tol = tol

    def _update_centroids(self, X, y):
        for i, c in enumerate(np.unique(y)):
            new_center = np.mean(X[y==c, :], axis=0)
            self.centroids[i] = new_center

    def _predict(self, x):
        return np.argmin(np.linalg.norm(self.centroids-x, axis=1))

    def predict(self, X):
        return np.apply_along_axis(lambda x: self._predict(x), 1, X)

    def fit(self, X):
        if self.centroids == 'random' or self.centroids is None:
            idx = np.random.randint(X.shape[0], size=self.n_clusters)
            self.centroids = X[idx, :]
            # print(self.centroids)

        for i in range(self.max_iter):
            y = self.predict(X)
            previous_centroids = self.centroids.copy()
            self._update_centroids(X, y)
```

```
                if np.all(np.linalg.norm(previous_centroids-self.centroids,
                                         axis=1)
                          < self.tol):

                    print('convergence at iteration-{}'.format(i))
                    break
            return self
```

[7]
```
init = iris_X[[0, 50, 100], :]
init
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [7. , 3.2, 4.7, 1.4],
       [6.3, 3.3, 6. , 2.5]])
```

[8]
```
model = MyKmeans(init=init).fit(iris_X)
```

```
convergence at iteration-3
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:21:
FutureWarning: elementwise comparison failed; returning scalar instead,
but in the future will perform elementwise comparison
```

(a) print out the coordinates of the cluster centers

[9]
```
model.centroids
```

```
array([[5.006     , 3.418     , 1.464     , 0.244     ],
       [5.9016129 , 2.7483871 , 4.39354839, 1.43387097],
       [6.85      , 3.07368421, 5.74210526, 2.07105263]])
```

(b) The number of members (sample points) in each cluster.

[10]
```
np.bincount(model.predict(iris_X))
```

```
array([50, 62, 38])
```

(c) According to the labels of data samples, how many of them are placed in wrong clusters? Use a majority vote to determine the label of each cluster.

[11]
```
error = iris_y - model.predict(iris_X)
print('There are %d samples are placed in wrong clusters.'
      %(error[error != 0].shape[0]))
```

```
There are 16 samples are placed in wrong clusters.
```

```
[12]  accuracy_score(iris_y, model.predict(iris_X))
```

```
0.8933333333333333
```

3 . We know that the GMM can be viewed as a “soft” clustering method. To simplify the difficulty level, we will implement the univariate GMM. Use the third feature (petal length) as the input to your GMM.

The settings are three Gaussians with the following initial values: $\mu_1 = 1, \mu_2 = 4, \mu_3 = 6, \sigma_1^2 = \sigma_2^2 = \sigma_3^2 = 1, a_1 = 0.5, a_2 = a_3 = 0.25$.

To have a unique answer, iterate the EM steps 3,000 times (epochs).

```
[13]  u = [1, 4, 6]
      stddev = [1, 1, 1]
      size = [0.5, 0.25, 0.25]
```

```
[14]  class MyGMM:

          def __init__(self, mean, stddev, size, epoch=3000):
              self.u = np.array(mean)
              self.cov = np.array(stddev)**2
              self.size = np.array(size)
              self.epoch = epoch


          def _E_step(self, X):
              '''
              get r_ic
              '''
              R = norm.pdf(X, self.u, self.cov)
              R = R*self.size
              R /=  (np.sum(R, axis=1).reshape(-1,1))
              return R

          def _M_step(self, R, X):
              Mc = np.sum(R, axis=0)
              new_size = Mc/R.shape[0]
```

```
        new_mean = np.array([np.sum(rc*X.reshape(-1))/mc for rc, mc in zi
        new_cov  = np.array([np.cov(rc*X.reshape(-1))    for rc, mc in zi
        return new_mean, new_cov, new_size

    def _update(self, old, new):
        if any(np.isnan(new)):
            return (old, True)

        return (old, True) if all(np.isclose(old, new)) else (new, False)

    def fit(self, X):
        for i in range(self.epoch):

            R = self._E_step(X)
            new_mean, new_cov, new_size = self._M_step(R, X)
            #print(i,new_mean, new_cov, new_size)

            self.u, is_close_1 = self._update(self.u, new_mean)
            self.cov, is_close_2 = self._update(self.cov, new_cov)
            self.size, is_close_3 = self._update(self.size, new_size)

            if all([is_close_1, is_close_2, is_close_3]):
                break
        return self

    def predict(self, X):
        R = self._E_step(X)
        return np.argmax(R, axis=1)
```

```
[15]  third_X = iris_X[:, 2].reshape(-1, 1)
      model = MyGMM(u, stddev, size).fit(iris_X[:, 2].reshape(-1, 1))
```

a. Print out the GMM parameters.

```
[16]  print('mean  ', model.u)
      print('stddev', model.cov**0.5)
      print('size  ', model.size)
```

```
mean   [1.46168703 4.83405382 4.8341634 ]
stddev [0.66605551 1.19305719 1.19306461]
size   [0.31889309 0.34055867 0.34054824]
```

b. If you want to convert the “soft” clustering results to “hard” clustering ones, how do you do it?

ANS: Using np.argmax over r_{ic}

```
def predict(self, X):
    R = self._E_step(X)
    return np.argmax(R, axis=1)
```

```
[17]  accuracy_score(iris_y, model.predict(third_X))
```

```
0.8333333333333334
```

c. Use your method in (b) to find the number of members in each cluster.

```
[18]  np.bincount(model.predict(third_X))
```

```
array([50, 75, 25])
```

4 . We used the play/no play example in the lecture. You are required to write a program to compute the C 4.5 decision tree with the “play/no play” data given in the PPT file.

Plot the computed decision tree.

In this problem, you need to convert continuous variables of temperature and humidity to discrete values according to the rules given in the PPT file.

```
[19]  feature_names = ['outlook', 'temp', 'humidity', 'windy']
```

```
[20]  data_y = np.array([0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0])
```

```
[21]  data_X = np.array([['sunny',    85, 85, 0],
                        ['sunny',    80, 90, 1],
                        ['overcast', 83, 78, 0],
                        ['rain',     70, 96, 0],
                        ['rain',     68, 80, 0],
                        ['rain',     65, 70, 1],
                        ['overcast', 64, 65, 1],
                        ['sunny',    72, 95, 0],
                        ['sunny',    69, 70, 0],
                        ['rain',     75, 80, 0],
                        ['sunny',    75, 70, 1],
                        ['overcast', 72, 90, 1],
                        ['overcast', 81, 75, 0],
                        ['rain',     71, 80, 1]])
```

[22]
```
temper_mask = data_X[:, 1].astype('int')
data_X[:, 1][temper_mask >= 80] = 'hot'
data_X[:, 1][(temper_mask >= 70) & (temper_mask <= 79)] = 'sweet'
data_X[:, 1][temper_mask < 70] = 'cold'
```

[23]
```
humidity_mask = data_X[:, 2].astype('int')
data_X[:, 2][humidity_mask >= 76] = 'high'
data_X[:, 2][humidity_mask < 76] = 'low'
```

[24]
```
data_X
```

```
array([['sunny', 'hot', 'high', '0'],
       ['sunny', 'hot', 'high', '1'],
       ['overcast', 'hot', 'high', '0'],
       ['rain', 'sweet', 'high', '0'],
       ['rain', 'cold', 'high', '0'],
       ['rain', 'cold', 'low', '1'],
       ['overcast', 'cold', 'low', '1'],
       ['sunny', 'sweet', 'high', '0'],
       ['sunny', 'cold', 'low', '0'],
       ['rain', 'sweet', 'high', '0'],
       ['sunny', 'sweet', 'low', '1'],
       ['overcast', 'sweet', 'high', '1'],
       ['overcast', 'hot', 'low', '0'],
       ['rain', 'sweet', 'high', '1']], dtype='<U8')
```

[25]
```
def entropy(S):
    n = S.shape[0]
    En = 0
    for u, count in zip(*np.unique(S, return_counts=True)):
        En += -1*(count/n)*np.log2(count/n)
    return En
#entropy(data_y)
```

[26]
```
def gain(S, T):
    En = entropy(S)
    n = S.shape[0]
    for u, count in zip(*np.unique(T, return_counts=True)):
        En -= (count/n)*entropy(S[T == u])
    return En

# for i, name in enumerate(feature_names):
#     print('gain for "%10s": %f'%(name, gain(data_y, data_X[:, i])))
```

```
[27]  def split_info(S, T):
          spt_info = 0
          n = S.shape[0]
          for u, count in zip(*np.unique(T, return_counts=True)):
              spt_info += -1*(count/n)*np.log2(count/n)
          return spt_info

      # for i, name in enumerate(feature_names):
      #     print('split_info for "%10s": %f'%(name, split_info(data_y, data_X[
```

```
[28]  def gain_ratio(S, T):
          return gain(S, T)/split_info(S, T)

      # for i, name in enumerate(feature_names):
      #     print('gain_ratio for "%10s": %f'%(name, gain_ratio(data_y, data_X[
```

```
[29]  def best_splitting_feature(X, features, y):
          gr = [gain_ratio(y, X[:, i]) for i, name in enumerate(features)]
          return np.argmax(gr)

      #best_splitting_feature(data_X, feature_names, data_y)
```

```
[30]  def create_leaf(target_values):

          leaf = {'splitting_feature' : None,
                  'splitting_feature_idx': None,
                  'sub_tree' : None,
                  'is_leaf':True }

          num_ones = len(target_values[target_values == 1])
          num_zero = len(target_values[target_values == 0])

          leaf['prediction'] =  int(num_ones > num_zero)

          return leaf
```

```
[31]  def decision_tree_create(X, features, y, current_depth=0, max_depth=10):
          remaining_features = features.copy()

          print ("----------------------------------------------------------")
          print ("Subtree, depth = %s (%s data points)." % (current_depth,
                                                             len(y)))



          '''
          if all Examples are positive or negative,
              return the single-node tree Root, with label=1 or 0
          '''
          if  len(np.unique(y)) == 1:
              print ("Stopping condition 1 & 2 reached."  )
```

```
            return create_leaf(y)


        # Additional stopping condition (limit tree depth)
        if current_depth >= max_depth:
            print ("Reached maximum depth. Stopping for now.")
            return create_leaf(y)

        # Find the best splitting feature
        # (recall the function best_splitting_feature implemented above)
        best_feature_idx = best_splitting_feature(X, features, y)
        best_feature_name = remaining_features[best_feature_idx]

        # Split on the best feature that we found.
        sub_tree_data = []
        unigue = np.unique(X[:, best_feature_idx])
        for u in unigue:
            mask = X[:, best_feature_idx] == u
            each_tree_X = np.delete(X[mask], best_feature_idx, axis=1)
            each_tree_y = y[mask]
            if each_tree_X.shape[0] > 0:
                sub_tree_data.append({'X':each_tree_X, 'y':each_tree_y})

        print ("Split on feature %s. (%s)"%(best_feature_name,
                                            len(sub_tree_data)))

        remaining_features.remove(best_feature_name)



        # Create a leaf node if the split is "perfect"
        if len(sub_tree_data) == 1:
            print ("Creating leaf node.")
            return create_leaf(sub_tree[0]['y'])

        # Repeat (recurse) on each subtrees
        sub_tree = {u:decision_tree_create(each_sub['X'],
                                           remaining_features, each_sub['y'],
                                           current_depth+1, max_depth)

                    for u, each_sub in zip(unigue, sub_tree_data)}

        return {'is_leaf'          : False,
                'prediction'       : None,
                'splitting_feature': best_feature_name,
                'splitting_feature_idx':best_feature_idx,
                'sub_tree' : sub_tree}
```

```
[32]:   def count_nodes(tree):
            if tree['is_leaf']:
                return 1
            return 1 + sum([count_nodes(sub) for _, sub in tree['sub_tree'].items
```

預設深度使用1，比照ppt example

```
[33] decision_tree = decision_tree_create(data_X,
                                         feature_names,
                                         data_y,
                                         max_depth=1)
```

```
--------------------------------------------------------------------
Subtree, depth = 0 (14 data points).
Split on feature outlook. (3)
--------------------------------------------------------------------
Subtree, depth = 1 (4 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 1 (5 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 1 (5 data points).
Reached maximum depth. Stopping for now.
```

```
[34] count_nodes(decision_tree)
```

```
4
```

Plotting the computed decision tree

```
[35] decision_tree
```

```
{'is_leaf': False,
 'prediction': None,
 'splitting_feature': 'outlook',
 'splitting_feature_idx': 0,
 'sub_tree': {'overcast': {'splitting_feature': None,
   'splitting_feature_idx': None,
   'sub_tree': None,
   'is_leaf': True,
   'prediction': 1},
  'rain': {'splitting_feature': None,
   'splitting_feature_idx': None,
   'sub_tree': None,
   'is_leaf': True,
   'prediction': 1},
  'sunny': {'splitting_feature': None,
   'splitting_feature_idx': None,
   'sub_tree': None,
   'is_leaf': True,
   'prediction': 0}}}
```

Test tree

```
[36] def classify(tree, x, annotate=False):
         # if the node is a leaf node.
         if tree['is_leaf']:
             if annotate:
                 print ("At leaf, predicting %s" % tree['prediction'])
             return tree['prediction']
         else:
             # split on feature.
             split_feature_name = tree['splitting_feature']
             split_feature_idx = tree['splitting_feature_idx']
             feature_value = x[split_feature_idx]

             if annotate:
                 print ("Split on %s = %s" % (split_feature_name,
                                              x[split_feature_idx]))

             #cut feature
             new_x = np.delete(x, split_feature_idx, axis=0)

             return classify(tree['sub_tree'][feature_value], new_x, annotate)
```

```
[37] print ('Play? ANS: %s ' % (bool(classify(decision_tree,
                                              ['sunny', 'sweet', 'low', '0'],
                                              annotate=True))))
```

```
Split on outlook = sunny
At leaf, predicting 0
Play? ANS: False
```

```
[38] print ('Play? ANS: %s ' % (bool(classify(decision_tree,
                                              ['rain', 'sweet', 'low', '0'],
                                              annotate=True))))
```

```
Split on outlook = rain
At leaf, predicting 1
Play? ANS: True
```

5 . Based on your C4.5 program on problem 4, revise it to accept continuous values of

temperature and humidity.

Inside your program, there must be a routine to convert each continuous number into three values, namely, low, mid, and high, based on maximizing gains.

(a) Use a pseudo code to explain how to perform the computation.

Step 1: list all sorted values in training set. EX: Humidity in the working problem.

Step 2: remove redundancy.

Step 3: Let the humidity be partitioned as {H1 <= H <= H0}, {H1 > H} and {H > H0}, where H0 H1 are number in the list in step 2.

Step 4: Compute all Gains based on all possible H0 and H1.

Step 5: Pick H0 and H2 with max Gain.

(b) Run your program to print out the conversion rules (such as temperature greater than xx is hot)

```
[39]  continuous_data_X = np.array([['sunny',    85, 85, 0],
                                    ['sunny',    80, 90, 1],
                                    ['overcast', 83, 78, 0],
                                    ['rain',     70, 96, 0],
                                    ['rain',     68, 80, 0],
                                    ['rain',     65, 70, 1],
                                    ['overcast', 64, 65, 1],
                                    ['sunny',    72, 95, 0],
                                    ['sunny',    69, 70, 0],
                                    ['rain',     75, 80, 0],
                                    ['sunny',    75, 70, 1],
                                    ['overcast', 72, 90, 1],
                                    ['overcast', 81, 75, 0],
                                    ['rain',     71, 80, 1]])
```

```
[40]  temperature = continuous_data_X[:, 1].astype('int')
      humidity    = continuous_data_X[:, 2].astype('int')
```

```
[41]  temperature
```

```
array([85, 80, 83, 70, 68, 65, 64, 72, 69, 75, 75, 72, 81, 71])
```

```
[42]  def chunk(X, H0, H1):
          result = []
          for i in X:
              if i > H1:
                  result.append('high'.format(H1))
              elif i <= H0:
                  result.append('low'.format(H0))
              else:
                  result.append('mid'.format(H0))
          return np.array(result)
```

```
[43]  def discretize(X, y):
          result = []
          unique = np.unique(X)
          H_pair = list(itertools.combinations(unique, 2))
          for pair in H_pair:
              result.append((pair, gain(y, chunk(X, pair[0], pair[1]))))
          return max(result, key=lambda x:x[1])
```

```
[44]  discretize(temperature, data_y)
```

```
((80, 83), 0.19726714791298539)
```

```
[45]  discretize(humidity, data_y)
```

```
((80, 95), 0.21721788321248015)
```

```
[46]  continuous_data_X[:, 1] = chunk(continuous_data_X[:, 1].astype('int') , 8
      continuous_data_X[:, 2] = chunk(continuous_data_X[:, 2].astype('int') , 8
```

```
[47]  continuous_data_X
```

```
array([['sunny', 'high', 'mid', '0'],
       ['sunny', 'low', 'mid', '1'],
       ['overcast', 'mid', 'low', '0'],
       ['rain', 'low', 'high', '0'],
       ['rain', 'low', 'low', '0'],
       ['rain', 'low', 'low', '1'],
       ['overcast', 'low', 'low', '1'],
       ['sunny', 'low', 'mid', '0'],
       ['sunny', 'low', 'low', '0'],
       ['rain', 'low', 'low', '0'],
       ['sunny', 'low', 'low', '1'],
       ['overcast', 'low', 'mid', '1'],
       ['overcast', 'mid', 'low', '0'],
       ['rain', 'low', 'low', '1']], dtype='<U8')
```

```
[48]  decision_tree = decision_tree_create(continuous_data_X,
                                             feature_names,
                                             data_y,
                                                max_depth=10)
```

```
--------------------------------------------------------------------
Subtree, depth = 0 (14 data points).
Split on feature temp. (3)
--------------------------------------------------------------------
Subtree, depth = 1 (1 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 1 (11 data points).
Split on feature humidity. (3)
--------------------------------------------------------------------
Subtree, depth = 2 (1 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 2 (7 data points).
Split on feature windy. (2)
--------------------------------------------------------------------
Subtree, depth = 3 (3 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 3 (4 data points).
Split on feature outlook. (3)
--------------------------------------------------------------------
Subtree, depth = 4 (1 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 4 (2 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 4 (1 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 2 (3 data points).
Split on feature outlook. (2)
--------------------------------------------------------------------
Subtree, depth = 3 (1 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 3 (2 data points).
Stopping condition 1 & 2 reached.
--------------------------------------------------------------------
Subtree, depth = 1 (2 data points).
Stopping condition 1 & 2 reached.
```

```
[49] count_nodes(decision_tree)
```

```
14
```

(c) draw the decision tree.

```
[50]   decision_tree
```

```
{'is_leaf': False,
 'prediction': None,
 'splitting_feature': 'temp',
 'splitting_feature_idx': 1,
 'sub_tree': {'high': {'splitting_feature': None,
   'splitting_feature_idx': None,
   'sub_tree': None,
   'is_leaf': True,
   'prediction': 0},
  'low': {'is_leaf': False,
   'prediction': None,
   'splitting_feature': 'humidity',
   'splitting_feature_idx': 1,
   'sub_tree': {'high': {'splitting_feature': None,
     'splitting_feature_idx': None,
     'sub_tree': None,
     'is_leaf': True,
     'prediction': 1},
    'low': {'is_leaf': False,
     'prediction': None,
     'splitting_feature': 'windy',
     'splitting_feature_idx': 1,
     'sub_tree': {'0': {'splitting_feature': None,
       'splitting_feature_idx': None,
       'sub_tree': None,
       'is_leaf': True,
       'prediction': 1},
      '1': {'is_leaf': False,
       'prediction': None,
       'splitting_feature': 'outlook',
       'splitting_feature_idx': 0,
       'sub_tree': {'overcast': {'splitting_feature': None,
         'splitting_feature_idx': None,
         'sub_tree': None,
         'is_leaf': True,
         'prediction': 1},
        'rain': {'splitting_feature': None,
         'splitting_feature_idx': None,
         'sub_tree': None,
         'is_leaf': True,
         'prediction': 0},
        'sunny': {'splitting_feature': None,
         'splitting_feature_idx': None,
         'sub_tree': None,
         'is_leaf': True,
         'prediction': 1}}}}},
    'mid': {'is_leaf': False,
     'prediction': None,
     'splitting_feature': 'outlook',
     'splitting_feature_idx': 0,
     'sub_tree': {'overcast': {'splitting_feature': None,
       'splitting_feature_idx': None,
       'sub_tree': None,
       'is_leaf': True,
       'prediction': 1},
      'sunny': {'splitting_feature': None,
       'splitting_feature_idx': None,
       'sub_tree': None,
       'is_leaf': True,
       'prediction': 0}}}}},
  'mid': {'splitting_feature': None,
   'splitting_feature_idx': None,
   'sub_tree': None,
   'is_leaf': True,
   'prediction': 1}}}
```

Test tree

```
[51]  print ('Play? ANS: %s ' % (bool(classify(decision_tree,
                                             ['rain', 'low', 'low', '1'],
                                             annotate=True))))
```

```
Split on temp = low
Split on humidity = low
Split on windy = 1
Split on outlook = rain
At leaf, predicting 0
Play? ANS: False
```

```
[52]  print ('Play? ANS: %s ' % (bool(classify(decision_tree,
                                             ['sunny', 'low', 'low', '1'],
                                             annotate=True))))
```

```
Split on temp = low
Split on humidity = low
Split on windy = 1
Split on outlook = sunny
At leaf, predicting 1
Play? ANS: True
```