Homework 2 (40 Pts)

STAT 451: Intro to Machine Learning (Fall 2020)
Instructor: Sebastian Raschka (sraschka@wisc.edu)

Course website: http://pages.stat.wisc.edu/~sraschka/teaching/stat479-fs2019/ (http://pages.stat.wisc.edu/~sraschka/teaching/stat479-fs2019/)

Due: Nov 2, (5:00 pm).

How to submit

As mentioned in the lecture, you need to send the `.ipynb` file with your answers plus an `.html` file, which will serve as a backup for us in case the `.ipynb` file cannot be opened on my or the TA's computer. In addition, you may also export the notebook as PDF and upload it as well.

The homework solution should be uploaded on Canvas. You can submit it as often as you like before the deadline.

Note that there are 13 tasks, and the HW is worth 78 pts in total (13*6pts=78pts).

Important

- The cells that require your code answer are marked with `"# YOUR CODE"` comments.
- Note that you may use 1 or more line of code for replacing each `"# YOUR CODE"` comment.

For example, imagine there is a question asking you to implement a threshold function that should return 1 if the input `x` is greater than 0.5 and otherwise. This could appear as follows in the the exercise:

```
def threshold_func(x):
    # YOUR CODE
```

A valid answer could be

```
def threshold_func(x):
    if x > 0.5:
        return 1
    else:
        return 0
```

Another valid solution could be

```
def threshold_func(x):
    return int(x > 0.5)
```

In [2]:
```
%load_ext watermark
%watermark  -d -u -a "Roshan Poduval" -v -p numpy,scipy,matplotlib,sklearn
```

```
Roshan Poduval
last updated: 2020-10-29

CPython 3.8.3
IPython 7.16.1

numpy 1.18.5
scipy 1.5.0
matplotlib 3.2.2
sklearn 0.23.1
```

In [3]:
```
import numpy as np
```

1) Implementing a "CART" Decision Tree from Scratch

In this first part of the homework, you are going to implement the CART decision tree algorithm we discussed in class. This decision tree algorithm will construct a binary decision tree based on maximizing Information Gain using the Gini Impurity measure on continuous features.

Implementing machine learning algorithms from scratch is a very important skill, and this homework will provide exercises that will help you to develop this skill. Even if you are interested in the more theoretical aspects of machine learning, being comfortable with implementing and trying out algorithms is vital for doing research, since even the more theoretical papers in machine learning are usually accompanied by experiments or simulations to a) verify results and b) to compare algorithms with the state-of-the art.

Since many students are not expert Python programmers (yet), I will provide partial solutions to the homework tasks such that you have a framework or guide to implement the solutions. Areas that you need to fill in will be marked with comments (e.g., `# YOUR CODE`). For these partial solutions, I first implemented the functions myself, and then I deleted parts you need to fill in by these comments. However, note that you can, of course, use more or fewer lines of code than I did. In other words, all that matter is that the function you write can create the same outputs as the ones I provide. How many lines of code you need to implement that function, and how efficient it is, does not matter here. The expected outputs for the respective functions will be provided for most functions so that you can double-check your solutions.

1.1) Splitting a node (4 pts)

First, we are going to implement a function that splits a dataset along a feature axis into sub-datasets. For this, we assume that the feature values are continuous (we are expecting NumPy float arrays). If the input is a NumPy integer array, we could convert it into a float array via

```
float_array = integer_array.astype(np.float64)
```

To provide an intuitive example of how the splitting function should work, suppose you are given the following NumPy array with four feature values, feature values 0-3:

```
np.array([0.0, 1.0, 4.0, 1.0, 0.0, 3.0, 1.0, 0.0, 1.0, 2.0])
```

The function you are going to implement should return a dictionary, where the dictionary key stores the information about which data point goes to the left child note and which data point goes to the right child node after applying a threshold for splitting.

For example, if we were to use a `split` function on the array shown above with a theshold $t = 2.5$, the split function should return the following dictionary:

```
{
 'left': array([0, 1, 3, 4, 6, 7, 8, 9]),   # smaller than or equal to threshold
 'right': array([2, 5])                     # larger than threshold'
 'threshold': 2.5                           # threshold for splitting, e.g., 2.5 me
ans <= 2.5
 }
```

Note that we also store a "threshold" key here to keep track of what value we used for the split -- we will need this later.

Now it's your turn to implement the split function.

In [4]:
```
# EDIT THIS CELL (4 pts)



def split(array, t):
    """
    Function that splits a feature based on a threshold.

    Parameters
    ----------
    array : NumPy array, type float, shape=(num_examples,)
      A NumPy array containing feature values (float values).

    t : float
      A threshold parameter for dividing the examples into
      a left and a right child node.

    Returns
    --------
    d : dictionary of the split
      A dictionary that has 3 keys, 'left', 'right', 'threshold'.
      The 'threshold' simply references the threshold t we provided
      as function argument. The 'left' child node is an integer array
      containing the indices of the examples corresponding to feature
      values with value <= t. The 'right' child node is an integer array
      stores the indices of the examples for which the feature value > t.
    """
    left = np.where(array<=t)
    right = np.where(array>t)
    d = {'left': left, 'right': right, 'threshold': t}
    return d
```

In [16]:
```
# DO NOT EDIT OR DELETE THIS CELL

ary = np.array([0.0, 1.0, 4.0, 1.0, 0.0, 3.0, 1.0, 0.0, 1.0, 2.0])

print(split(ary, t=2.5))

print(split(ary, t=1.5))

print(split(ary, t=-0.5))

print(split(ary, t=1.0))
```

```
{'left': array([0, 1, 3, 4, 6, 7, 8, 9]), 'right': array([2, 5]), 'threshol
d': 2.5}
{'left': array([0, 1, 3, 4, 6, 7, 8]), 'right': array([2, 5, 9]), 'threshol
d': 1.5}
{'left': array([], dtype=int64), 'right': array([0, 1, 2, 3, 4, 5, 6, 7, 8,
9]), 'threshold': -0.5}
{'left': array([0, 1, 3, 4, 6, 7, 8]), 'right': array([2, 5, 9]), 'threshol
d': 1.0}
```

1.2) Implement a function to compute the Gini Impurity (4 pts)

After implementing the splitting function, the next step is to implement a criterion function so that we can compare splits on different features. I.e., we use this criterion function to decide which feature is the best feature to split for growing the decision tree at each node. As discussed in class, our splitting criterion will be Information Gain. However, before we implement an Information Gain function, we need to implement a function that computes the Gini Impurity at each node, which we need to compute Information Gain. For your reference, we defined Gini Impurity as follows:

$$G(p) = 1 - \sum_i (p_i)^2$$

where you can think of p_i as the proportion of examples with class label i at a given node.

In [5]:
```
# EDIT THIS CELL  (4 pts)


def gini(array):
    """
    Function that computes the Gini Impurity.

    Parameters
    -----------
    array : NumPy array, type int, shape=(num_examples,)
      A NumPy array containing integers representing class
      labels.

    Returns
    ----------
    Gini impurity (float value).

    Example
    ----------
    >>> gini(np.array([0, 0, 1, 1]))
    0.5

    """
    if len(array) == 0:
        return 0

    n = len(array)
    gini = 1
    for i in range(np.max(np.unique(array)) + 1):
        gini = gini - (sum(array==i)/n)**2

    return gini
```

TIP: To check your solution, try out the `gini` function on some example arrays. Note that the Gini impurity is maximum (0.5) if the classes are uniformly distributed; it is minimum if the array contains labels from only one single class.

In [6]:
```
# DO NOT EDIT OR DELETE THIS CELL

print(round(gini(np.array([0, 1, 0, 1, 1, 0])), 4))
print(round(gini(np.array([1, 2])), 4))
print(round(gini(np.array([1, 1])), 4))
print(round(gini(np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])), 4))
print(round(gini(np.array([0, 0, 0])), 4))
print(round(gini(np.array([1, 1, 1, 0, 1, 4, 4, 2, 1])), 4))
```

```
0.5
0.5
0.0
0.1653
0.0
0.6173
```

1.3) Implement Information Gain (4 pts)

Now that you have a working solution for the `entropy` function, the next step is to compute the Information Gain. For your reference, information gain is computed as

$$GAIN(\mathcal{D}, x_j) = H(\mathcal{D}) - \sum_{v \in Values(x_j)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} H(\mathcal{D}_v).$$

In [6]:
```
# EDIT THIS CELL (4 pts)


def information_gain(x_array, y_array, split_dict):
    """
    Function to compute information gain.

    Parameters
    -----------

    x_array : NumPy array, shape=(num_examples)
      NumPy array containing the continuous feature
      values of a given feature variable x.

    y_array : NumPy array, shape=(num_examples)
      NumPy array containing the integer class labels
      corresponding to the training examples.

    split_dict : dictionary
      A dictionary created by the `split` function, which
      contains the indices for the left and right child node.

    Returns
    ---------

    Information gain for the given split in `split_dict`.

    """
    parent_n = len(x_array)
    parent_entropy = gini(y_array)

    for child in ('left', 'right'):
        # TIP: freq := |D_v| / |D|
        freq = len(y_array[split_dict[child]])/parent_n
        child_entropy = gini(y_array[split_dict[child]])
        child_entropy = gini(y_array[split_dict[child]])
        parent_entropy -= freq * child_entropy

    return parent_entropy
```

I added the following code cell for your convenience to double-check your solution. If your results don't match the results shown below, there is a bug in your implementation of the `information_gain` function.

In [8]:
```
# DO NOT EDIT OR DELETE THIS CELL

x_ary = np.array([0.0, 1.0, 4.0, 1.0, 0.0, 3.0, 1.0, 0.0, 1.0, 2.0])
y_ary = np.array([0, 1, 1, 0, 0, 0, 1, 1, 0, 0])

split_dict_1 = split(ary, t=2.5)
print(information_gain(x_array=x_ary,
                       y_array=y_ary,
                       split_dict=split_dict_1))

split_dict_2 = split(ary, t=1.5)
print(information_gain(x_array=x_ary,
                       y_array=y_ary,
                       split_dict=split_dict_2))

split_dict_3 = split(ary, t=-1.5)
print(information_gain(x_array=x_ary,
                       y_array=y_ary,
                       split_dict=split_dict_3))
```

```
0.004999999999999977
0.003809523809523735
0.0
```

1.4) Creating different splitting thresholds (4 pts)

Now, we should have almost all the main components that we need for implementing the CART decision tree algorithm: a `split` function, a `gini` function, and an `information_gain` function based on the `gini` function. However, since we are working with continuous feature variables, we need to find a good threshold t on the number line of each feature, which we can use with our function `split`.

For simplicity, we are going to implement a function that creates different thresholds based on the values found in a given feature variable. More precisely, we are going to implement a function `get_thresholds` that returns the lowest and highest feature value in a feature value array, plus the midpoint between each adjacent pairs of features (assuming the feature variable is sorted).

For example, if a feature array consists of the values

```
[0.1, 1.2, 2.4, 2.5, 2.7, 3.3, 3.7]
```

the returned thresholds should be

```
[0.1, 0.1+1.2/2, 1.2+2.4/2, 2.4+2.5/2, 2.5+2.7/2, 2.7+3.3/2, 3.7]
```

In [7]:
```
# EDIT THIS CELL  (4 pts)

def get_thresholds(array):
    """
    Get thresholds from a feature array.

    Parameters
    -----------
    array : NumPy array, type float, shape=(num_examples,)
      Array with feature values.

    Returns
    -----------
    NumPy float array containing thresholds.

    """
    # YOUR CODE (multiple lines)
    n = len(array)
    array = np.sort(array)
    output = np.zeros(n+1)
    for i in range(n):
        if i == 0:
            output[i] = array[i]
        else:
            output[i] = (array[i-1] + array[i]) / 2.0

    output[n] = array[n-1]

    return output
```

In [10]:
```
# DO NOT EDIT OR DELETE THIS CELL

a = np.array([0.1, 1.2, 2.4, 2.5, 2.7, 3.3, 3.7])
print(get_thresholds(a))

b = np.array([3.7, 2.4, 1.2, 2.5, 3.3, 2.7, 0.1])
print(get_thresholds(b))
```

```
[0.1  0.65 1.8  2.45 2.6  3.   3.5  3.7 ]
[0.1  0.65 1.8  2.45 2.6  3.   3.5  3.7 ]
```

1.5) Selecting the best splitting threshold (4 pts)

In the previous section, we implemented a function `get_thresholds` to create different splitting thresholds for a given feature. In this section, we are now implementing a function that selects the best threshold (the threshold that results in the largest information gain) from the array returned by `get_thresholds` by combining the

- `get_thresholds`
- `split`
- `information_gain`

functions.

In [8]:
```
# EDIT THIS CELL  (4 pts)

def get_best_threshold(x_array, y_array):
    """
    Function to obtain the best threshold
    based on maximizing information gain.

    Parameters
    ----------
    x_array : NumPy array, type float, shape=(num_examples,)
      Feature array containing the feature values of a feature
      for each training example.
    y_array : NumPy array, type int, shape=(num_examples,)
      NumPy array containing the class labels for each
      training example.


    Returns
    -----------
    A float representing the best threshold to split the given
    feature variable on.
    """

    all_thresholds = get_thresholds(x_array)
    info_gains = np.zeros(all_thresholds.shape[0])

    for idx, t in enumerate(all_thresholds):

        split_dict_t = split(x_array, t=t)
        ig = information_gain(x_array, y_array, split_dict_t)

        info_gains[idx] = ig

    best_idx = np.argmax(info_gains)
    best_threshold = all_thresholds[best_idx]

    return best_threshold
```

In [12]:
```
x_ary = np.array([0.0, 1.0, 4.0, 1.0, 0.0, 3.0, 1.0, 0.0, 1.0, 2.0])
y_ary = np.array([0, 1, 1, 0, 0, 0, 1, 1, 0, 0])

print(get_best_threshold(x_array=x_ary,
                         y_array=y_ary))

x_ary = np.array([0.0, 3.0, 1.0, 0.0, 1.0, 2.0, 0.0, 1.0, 4.0, 1.0,])
y_ary = np.array([0, 0, 1, 1, 0, 0, 0, 1, 1, 0])

print(get_best_threshold(x_array=x_ary,
                         y_array=y_ary))
```

```
3.5
3.5
```

1.6) Decision Tree Splitting (4 pts)

The next task is to combine all the previously developed functions to recursively split a dataset on its different features to construct a decision tree that separates the examples from different classes well. We will call this function `make_tree`.

For simplicity, the decision tree returned by the `make_tree` function will be represented by a Python dictionary. To illustrate this, consider the following dataset consisting of 6 training examples (class labels are 0 or 1) and 2 feature variables X_0 and X_1:

```
Inputs:
 [[0. 0.]
  [0. 1.]
  [1. 0.]
  [1. 1.]
  [2. 0.]
  [2. 1.]]

Labels:
 [0 1 0 1 1 1]
```

Based on this dataset with 6 training examples and two features, the resulting decision tree in form of the Python dictionary should look like as follows:

You should return a dictionary with the following form:

```
{'X_1 <= 0.000000': {'X_0 <= 1.500000': array([0, 0]),
                     'X_0 > 1.500000': array([1])
                    },

 'X_1 > 0.000000': array([1, 1, 1])
}
```

Let me further illustrate what the different parts of the dictionary mean. Here, the `'X_1'` in `'X_1 <= 0'` refers feature 2 (the first column of the NumPy array; remember that Python starts the index at 0, in contrast to R).

- 'X_1 <= 0': For training examples stored in this node where the 2nd feature is less than or equal to 0.
- 'X_1 > 0': For training examples stored in this node where the 2nd feature is larger than 0.

The "array" is a NumPy array that stores the class labels of the training examples at that node. In the case of `'X_1 <= 0'` we actually store actually a sub-dictionary, because this node can be split further into 2 child nodes with `'X_0 <= 1.500000'` and `'X_0 > 1.500000'`.

In [9]:
```
# EDIT THIS CELL (4 pts)

def make_tree(X, y):
    """
    A recursive function for building a binary decision tree.

    Parameters
    ----------
    X : NumPy array, type float, shape=(num_examples, num_features)
      A design matrix representing the feature values.

    y : NumPy array, type int, shape=(num_examples,)
      NumPy array containing the class labels corresponding to the training ex
amples.

    Returns
    ----------
    Dictionary representation of the decision tree.
    """

    # Return class label array if node is empty or pure (1 example in leaf nod
e)
    if np.all(y == y[0]):
        return y

    # Select the best threshold for each feature
    n_features = X.shape[1]
    thresholds = np.zeros(n_features) # YOUR CODE
    for i in range(n_features):
        thresholds[i] = get_best_threshold(X.T[i][:], y)

    # Compute information gain for each feature based on the best threshold fo
r each feature

    gains = np.zeros(X.shape[1])
    split_dicts = []

    for idx, (feature, threshold) in enumerate(zip(X.T, thresholds)):
        split_dict = split(feature, threshold)
        split_dicts.append(split_dict)
        ig = information_gain(feature, y, split_dict)
        gains[idx] = ig

    # Early stopping if there is no information gain
    if (gains <= 1e-05).all():
        return y

    # Else, get best feature
    best_feature_idx = np.argmax(gains) # YOUR CODE

    results = {}

    subset_dict = split_dicts[best_feature_idx]

    for node in ('left', 'right'):
        child_y_subset = y[subset_dict[node]]
```

```
        child_X_subset = X[subset_dict[node]]

        if node == 'left':
            results["X_%d <= %f" % (best_feature_idx, subset_dict['threshold'
])] = \
                    make_tree(child_X_subset, child_y_subset)

        else:
            results["X_%d > %f" % (best_feature_idx, subset_dict['threshold'
])] = \
                    make_tree(child_X_subset, child_y_subset)

    return results
```

I added the following code cell for your convenience to double-check your solution. If your results don't match the results shown below, there is a bug in your implementation of the `make_tree` function.

In [14]:
```
# DO NOT EDIT OR DELETE THIS CELL

x1 = np.array([0., 0., 1., 1., 2., 2.])
x2 = np.array([0., 1., 0., 1., 0., 1.])
X = np.array( [x1, x2]).T
y = np.array( [0,  1,  0,  1,  1,  1])

print('Inputs:\n', X)
print('\nLabels:\n', y)

print('\nDecision tree:\n', make_tree(X, y))
```

```
Inputs:
 [[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [2. 0.]
 [2. 1.]]

Labels:
 [0 1 0 1 1 1]

Decision tree:
 {'X_1 <= 0.000000': {'X_0 <= 1.500000': array([0, 0]), 'X_0 > 1.500000': arr
ay([1])}, 'X_1 > 0.000000': array([1, 1, 1])}
```

1.7) Building a Decision Tree API (4 pts)

The final step of this part of the homework is now to write an API around our decision tree code so that we can use is for making predictions. Here, we will use the common convention, established by scikit-learn, to implement the decision tree as a Python class with

- a `fit` method that learns the decision tree model from a training set via the `make_tree` function we already implemented;
- a `predict` method to predict the class labels of training examples or any unseen data points.

For making predictions, since not all leaf nodes are guaranteed to be single training examples, we will use a majority voting function to predict the class label as discussed in class. I already implemented a `_traverse` method, which will recursively traverse a decision tree dictionary that is produced by the `make_tree` function.

Note that for simplicity, the `predict` method will only be able to accept one data point at a time (instead of a collection of data points). Hence `x` is a vector of size $\mathbb{R}^m$, where m is the number of features. I use capital letters `X` to denote a matrix of size $\mathbb{R}^{n \times m}$, where n is the number of training examples.

In [19]:
```
# EDIT THIS CELL (4 pts)
from scipy import stats


class CARTDecisionTreeClassifer(object):

    def __init__(self):
        pass

    def fit(self, X, y):
        self.splits_ = make_tree(X, y)

    def _majority_vote(self, label_array):
        """ Returns a class label by majority voting
            on an array label_array
        """
        return stats.mode(label_array)[0][0] # YOUR CODE

    def _traverse(self, x, d):
        if isinstance(d, np.ndarray):
            return d
        for key in d:


            if '<=' in key:
                name, value = key.split(' <= ')
                feature_idx = int(name.split('_')[-1])
                value = float(value)
                if x[feature_idx] <= value:
                    return self._traverse(x, d[key])
            else:
                # YOUR CODE
                name, value = key.split(' > ')
                feature_idx = int(name.split('_')[-1])
                value = float(value)
                if x[feature_idx] > value:
                    return self._traverse(x, d[key])

    def predict(self, x):
        label_array = self._traverse(x, self.splits_)
        return self._majority_vote(label_array)
```

I added the following code cell for your convenience to double-check your solution. If your results don't match the results shown below, there is a bug in your implementation of the `make_tree` function.

In [20]:
```
# DO NOT EDIT OR DELETE THIS CELL

tree = CARTDecisionTreeClassifer()
tree.fit(X, y)

print(tree.predict(np.array([0., 0.])))
print(tree.predict(np.array([0., 1.])))
print(tree.predict(np.array([1., 0.])))
print(tree.predict(np.array([1., 0.])))
print(tree.predict(np.array([1., 1.])))
print(tree.predict(np.array([2., 0.])))
print(tree.predict(np.array([2., 1.])))
```

```
0
1
0
0
1
1
1
```

2) Bagging

In this second part of this homework, you will be combining multiple decision trees to a bagging classifier. This time, we will be using the decision tree algorithm implemented in scikit-learn (which is some variant of the CART algorithm for binary splits, as implemented earlier and discussed in class).

2.1 Bootrapping (2x 4pts)

As you remember, bagging relies on bootstrap sampling. So, as a first step, your task is to implement a function for generating bootstrap samples. In this exercise, for simplicity, we will perform the computations based on the Iris dataset.

On an interesting side note, scikit-learn recently updated their version of the Iris dataset since it was discovered that the Iris version hosted on the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Iris/ (https://archive.ics.uci.edu/ml/datasets/Iris/)) has two data points that are different from R. Fisher's original paper (Fisher,R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).) and changed it in their most recent version. Since most students may not have the latest scikit-learn version installed, we will be working with the Iris dataset that is deposited on UCI, which has become quite the standard in the Python machine learning community for benchmarking algorithms. Instead of manually downloading it, we will be fetching it through the `mlxtend` (http://rasbt.github.io/mlxtend/ (http://rasbt.github.io/mlxtend/)) library that you installed in the last homework.

In [25]:
```
# DO NOT EDIT OR DELETE THIS CELL

from mlxtend.data import iris_data
X, y = iris_data()

print('Number of examples:', X.shape[0])
print('Number of features:', X.shape[1])
print('Unique class labels:', np.unique(y))
```

```
Number of examples: 150
Number of features: 4
Unique class labels: [0 1 2]
```

Use scikit-learn's `train_test_split` function to divide the dataset into a training and a test set.

- The test set should contain 45 examples, and the training set should contain 105 examples.
- To ensure reproducible results, use `123` as a random seed.
- Perform a stratified split.

In [27]:
```
# EDIT THIS CELL (4 pts)


from sklearn.model_selection import train_test_split


X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=45, # YOUR CODE
                                                    train_size=105,
                                                    random_state=123,
                                                    stratify=y
                                                   )

print('Number of training examples:', X_train.shape[0])
print('Number of test examples:', X_test.shape[0])
```

```
Number of training examples: 105
Number of test examples: 45
```

Next we are implementing a function to generate bootstrap samples of the training set. In particular, we will perform the bootstrapping as follows:

- Create an index array with values 0, ..., 104.
- Draw a random sample (with replacement) from this index array using the `choice` method of a NumPy `RandomState` object that is passed to the function as `rng`.
- Select training examples from the X array and labels from the y array using the new sample of indices.

In [33]:
```
# EDIT THIS CELL  (4 pts)

def draw_bootstrap_sample(rng, X, y):
    sample_indices = np.arange(X.shape[0])
    bootstrap_indices = rng.choice(sample_indices, 105) # YOUR CODE
    return X[bootstrap_indices], y[bootstrap_indices]
```

I added the following code cell for your convenience to double-check your solution. If your results don't match the results shown below, there is a bug in your implementation of the `draw_bootstrap_sample` function.

In [20]:
```
# DO NOT EDIT OR DELETE THIS CELL

rng = np.random.RandomState(123)
X_boot, y_boot = draw_bootstrap_sample(rng, X_train, y_train)

print('Number of training inputs from bootstrap round:', X_boot.shape[0])
print('Number of training labels from bootstrap round:', y_boot.shape[0])
print('Labels:\n', y_boot)
```

```
Number of training inputs from bootstrap round: 105
Number of training labels from bootstrap round: 105
Labels:
 [0 0 1 0 0 1 2 0 2 1 0 0 2 1 1 1 1 2 1 1 2 0 2 1 2 1 1 1 0 1 0 0 1 2 0 0 0
 0 2 1 1 2 1 2 1 1 2 1 2 0 1 1 2 2 1 0 1 0 2 2 0 1 0 2 0 0 0 0 1 2 0 0 1 0
 1 1 0 1 1 2 2 0 2 0 2 0 1 1 2 2 0 2 2 2 0 1 0 1 2 2 2 1 0 0 0]
```

2.2 Baggging classifier from decision trees (4 pts)

In this section, you will implement a Bagging algorithm based on the `DecisionTreeClassifier`. I provided a partial solution for you.

In [40]:
```
# EDIT THIS CELL (4 pts)


from sklearn.tree import DecisionTreeClassifier


class BaggingClassifier(object):

    def __init__(self, num_trees=10, random_state=123):
        self.num_trees = num_trees
        self.rng = np.random.RandomState(random_state)


    def fit(self, X, y):
        self.trees_ = [DecisionTreeClassifier(random_state=self.rng) for i in
range(self.num_trees)]
        for i in range(self.num_trees):
            X_boot, y_boot = draw_bootstrap_sample(self.rng, X, y)
            # YOUR CODE to
            # fit the trees in self.trees_ on the bootstrap samples
            self.trees_[i] = self.trees_[i].fit(X_boot, y_boot)


    def predict(self, X):
        ary = np.zeros((X.shape[0], len(self.trees_)), dtype=np.int)
        for i in range(len(self.trees_)):
            ary[:, i] = self.trees_[i].predict(X)

        maj = np.apply_along_axis(lambda x:
                                  np.argmax(np.bincount(x)),
                                            axis=1,
                                            arr=ary)
        return maj
```

I added the following code cell for your convenience to double-check your solution. If your results don't match the results shown below, there is a bug in your implementation of the `BaggingClassifier()` .

In [22]:
```
# DO NOT EDIT OR DELETE THIS CELL

model = BaggingClassifier()
model.fit(X_train, y_train)

predictions = model.predict(X_test)

print('Individual Tree Accuracies:')
for tree in model.trees_:
    predictions = tree.predict(X_test)
    print('%.1f%%' % ((predictions == y_test).sum() / X_test.shape[0] * 100))

print('\nBagging Test Accuracy: %.1f%%' % ((predictions == y_test).sum() / X_t
est.shape[0] * 100))
```

```
Individual Tree Accuracies:
88.9%
93.3%
97.8%
93.3%
93.3%
93.3%
91.1%
97.8%
97.8%
97.8%

Bagging Test Accuracy: 97.8%
```