Python

Part 1: Scikit-learn Tutorial (5 Points)

You should first ensure that you have sklearn, numpy, and scipy installed. If you successfully completed the Data Exploration Assignment, you already have these installed.

1.1 Modules to import

We will be using the sklearn Decision Tree Classifier, Random Forest Classifier, and Naive Bayes Classifier. We will also use some sklearn model selection and evaluation tools, and the Winee Dataset. To use all of these tools, you should import the following:

- From the module `sklearn.tree`, you should import `DecisionTreeClassifier`
- From the module `sklearn.naive_bayes` you should import `BernoulliNB`
- From the module `sklearn.ensemble`, you should import `RandomForestClassifier`
- From the module `sklearn.model_selection` you should import `train_test_split`
- From the module `sklearn.metrics`, you should import `confusion_matrix` and `classification_report`
- From the module `sklearn.datasets`, you should import `load_wine`

1.2 Preparing the dataset (1 Point)

The Wine dataset can be used much like the Boston Housing dataset from the previous assignment. Use the function `train_test_split()` with a test size of .5 to split the `data` and `target` into four arrays: training data, testing data, training targets, and testing targets.

1.3 Creating the classifier

Create an instance of the `DecisionTreeClassifier` class and save it in some variable.

1.4 Training the classifier (1 Point)

Use the `fit()` method of the `DecisionTreeClassifier` to train the classifier using your training data and training targets.

1.5 Making predictions (1 Point)

Use the `predict()` method of the `DecisionTreeClassifier` to predict the targets for your testing data. Save these results in some variable.

1.6 Evaluation the classifiers (2 Points)

Repeat steps 1.3-1.5 for each classifier. Take care to save the results of each classifier for comparison. Additionally, for the Random Forest Classifier, access the `feature_importances` attribute and create a labeled bar chart with matplotlib.

Now that you know the predicted targets for the testing data, you can compare them with the known targets. Use the `confusion_matrix` and `classification_report` functions to compare the known targets with the predicted targets of each classifier. Using the output, explain what the precision and recall indicate about each classifier's performance.

Part 2: Decision Tree Functions (10 Points)

Recall Hunt's Algorithm for decision trees:

2.1 Hunt's Algorithm

1. The initial node t_1 contains all training records
2. Procedure:

- D_t is the set of training records in a node t
- if D_t contains records that belong the the same class y_t
  - t is a leaf node labeled as y_t
- if D_t is an empty set:
  - t is a leaf node labeled as the default class, y_d
- if D_t contains records that belong to more than one class
  - use an attribute test to split the data into smaller subsets and assign each subset to a new child node

3. recursively apply the procedure (2) to each node

You will solve some of the questions presented by Hunt's Algorithm:

- Entropy calculation
- Information gain calculation
- Deciding upon a split

To simplify your work, Part 2 only considers binary categorical features. There are also examples for each task, and test cases are included in the `test.py` file. To use these test cases, place `test.py` into the same directory as your assignmet, import the test functions into your assignment, and pass your functions as arguments to the test functions. Feel free to examine `tests.py` to help you.

Example:

```
>>>from tests import test_entropy, test_binary_split ...
>>>def entropy(y):
    # your function here
    return 13.37
>>>test_entropy(entropy)
True # (or maybe not True...)
```

2.2 Entropy Function (3 Points)

Entropy is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i)\log_2 P(x_i)$$

Following this, recall from the lecture and slides that the entropy at a given node t is defined as

$$Entropy(t) = -\sum_{j} P\left(\frac{j}{t}\right)\log_2 P\left(\frac{j}{t}\right)$$

where $P\left(\frac{j}{t}\right)$ is the relative frequency of a class j in node t.

Example: Say a node t contains the following training samples:

$feature1$	$feature2$	$label$
0	0	red
0	1	red
1	0	$blue$

Then, the classes are $\{red, blue\}$, the relative frequencies of the classes are $\frac{2}{3}$ and $\frac{1}{3}$, and the entropy is $-\left(\frac{2}{3}\log_2\frac{2}{3} + \frac{1}{3}\log_2\frac{1}{3}\right)$.

2.2.1 Your task

Write a function called `entropy(y)` that accepts a one-dimensional (with a shape of `(n,)`) NumPy `ndarray` as the only argument. This `ndarray` will contain the class labels of the training samples at a node. The function should compute the entropy at this node, and return it as a floating point value. The function `entropy(y)` will be tested by the provided function `test_entropy()`.

Example:

```
>>>y = np.array([0, 0, 0, 1, 2, 2])
>>>entropy(y)
1.4591
```

2.2.2 Hints

- The NumPy function `np.count_nonzero()` can be used to get the count of elements in a NumPy `ndarray`. Do not let the name deceive you: any values can be counted, not just nonzero values. Pass a boolean expression between an `ndarray` and a scalar as the argument, and the boolean expresisonwill be broadcast to the entire `ndarray`.

  Example:

  ```
  >>>import numpy as np
  >>>big_list = np.array([0, 1, 1, 0, 2, 3, 2, 3, 3, 1, 2, 3])
  >>>np.count_nonzero(big_list == 3)
  4
  >>>np.count_nonzero(big_list == 2)
  3
  >>>np.count_nonzero(big_list == 1)
  3
  >>>np.count_nonzero(big_list == 0)
  2
  ```

- The log function may be imported from the `math` package. Take care that you select $\log_2$ and not $\log_{10}$ or ln

---

2.3 Find each possible split at a node for a given feature (2 Points)

Recall from Hunt's algorithm that if a node t is impure, an attribute test should be used to determine which splits to make. We should perform the attribute test on each possible split for each possible feature to determine which split to make. In order to do this, we should first determine what are the possible splits. The attribute test will be computed for each split by another function you will write later in the assignment.

For example, if a node t contains the following training samples:

feature1	*feature2*	*label*	*index*
high	*no*	*red*	1
high	*maybe*	*red*	2
low	*yes*	*blue*	3
low	*yes*	*blue*	4
high	*no*	*blue*	5
low	*maybe*	*green*	6

then t could be split in many ways:

On feature 1 (binary categorical, so only a single binary split possible):

low	*high*
$\{3, 4, 6\}$	$\{1, 2, 5\}$

On feature 2 (3-nary categorical, so only a single 3-nary split is possible, but several binary splits are possible):

$$\begin{array}{ccc}
no & \neg no & \\
\{1,5\} & \{2,3,4,6\} & \\
maybe & \neg maybe & \\
\{2,6\} & \{1,3,4,5\} & \\
yes & \neg yes & \\
\{3,4\} & \{1,2,5,6\} & \\
no & maybe & yes \\
\{1,5\} & \{2,6\} & \{3,4\}
\end{array}$$

This can continue up to n-ary splits, where n is the number of categories of the feature. Similar splits can be made for all categorical features.

For this assignment, let us only consider decision trees where each feature is a binary categorical variable, so it is only necessary to find the single binary split for each feature. This will make your coding task significantly easier.

2.3.1 Your task

Write a function called `binary_split(X, feature)` that accepts

- X: A two-dimensional `ndarray` of unlabeled training samples (with a shape of `(n, m)` where `n` is the number of training samples and `m` is the number of features).
- feature: The index of a feature to split on. Since `m` is the number of features, `feature` $<$ `m` always.

The function should compute the single possible binary split of the training samples using the specified feature. Return this split as a tuple of list, with each set

1. corresponding to a child of the split. Since we are considering only binary categorical features, there will only be two children, and so two sets.
2. containing the indices of the training samples belonging to that child.

Example:

```
>>>X = np.array([[1, "A"], [1, "B"], [0, "A"]])
>>>binary_split(X, 0)
[{2}, {0, 1}]
>>>binary_split(X, 1)
[{1}, {0, 2}]
```

2.3.2 Hints

The function `np.where(condition)` can be used to get a tuple whose first element is an array containing all indices where the condition is true. Like in the `np.count_nonzero()` function, this condition can be broadcast to an array like this: `np.where(big_list==3)[0]` would return the `ndarray` with all the indices of `biglist` that have the value of 3. This function can also be combined with `ndarray` indexing: if `big_list` were two-dimensional, `np.where(big_list[:, 2]==3)[0]` would return the indices of rows in `big_list` where the column at index 2 has the value 3.

2.4 Information Gain (3 Points)

Recall that information gain is defined as the original entropy of a node minus the weighted sum of the entropies of the new child nodes. That is, for a parent node D being split into v many child nodes D_i:

$$InfoGain(D) = H(D) - \sum_{i=1}^{v} \frac{|D_i|}{|D|} H(D_i)$$

2.4.1 Your task

Write a function called `information_gain(X, y, feature)` that accepts

- X: A two-dimensional `ndarray` of unlabeled training samples (with a shape of `(n, m)` where `n` is the number of training samples and `m` is the number of features).
- y: A one-dimensional `ndarray` of labels (with a shape of `(n,)`).
- feature: The index of a feature to split on. Since `m` is the number of features, `feature` $<$ `m` always.

The function should compute the information gain of splitting the samples `X, y` on the feature `feature`. You should call your `binary_split()` function to get the indices for the split.

Example:

```
>>>X = np.array([[0, 2], [0, 3], [0, 3], [1, 2], [1, 3], [1, 3]])
>>>y = np.array([0, 1, 1, 0, 0, 1])
>>>information_gain(X, y, 0)
0.08170416594551044
>>>information_gain(X, y, 1)
0.4591479170272448
```

2.4.2 Hints

You may find it useful to convert a `set()` into a `list()`. This can be done with the `list()` function:

Example:

```
>>>s = {1, 2, 3, 5, 8}
>>>type(s)
set
>>>l = list(s)
>>>type(l)
list
```

2.5 Determine Best Split (2 Points)

Using the functions you have created above, you should be able to determine the best split for a node.

2.5.1 Your task

Write a function called `determine_best_split(X, y)` that accepts `X` and `y` in the same format as the `information_gain()` function.

The function should find the feature that maximizes information gain when split upon, and return the index of this feature. You should call your `information_gain()` function to calculate the information gain for each feature.

Example:

```
>>>X = np.array([[0, 2], [1, 2], [1, 3]])
>>>y = np.array([0, 0, 1])
>>>determine_best_split(X, y)
1
>>>y = np.array([0, 1, 1])
>>>determine_best_split(X, y)
0
```