Lab 6 (Part 1): Identification Trees

You are expected, but not required, to complete this portion of the lab by **11:59pm X**. The official, final deadline for both Part 1 and Part 2 is **11:59pm Thursday, October 29**.

Identification (or classification) trees are a powerful technique for classifying certain points based on common characteristics with other data points. In this section of the lab, we will first learn what it means to classify a data point. Then, we will use ID trees to classify certain points. Finally, we will learn how to construct new ID trees from raw data, using the techniques we learned in class.

Identification (or classification) trees are a powerful technique for classifying certain points based on common characteristics with other data points. In this section of the lab, we will first learn what it means to classify a data point. Then, we will use ID trees to classify certain points. Finally, we will learn how to construct new ID trees from raw data, using the techniques we learned in class.

Contents

Part 1A: Using an ID Tree to classify unknown points

A data point, conceptually, is a sample of information with multiple associated values.

We will represent a data point as a dictionary mapping attributes (features) to these values. For example, the following could represent a data point in our vampires example from lecture:

```
data_point_1 = {"name":"person1", "Vampire?":"Vampire", "Shadow":"?",
"Eats Garlic":"No", "Complexion":"Ruddy", "Accent":"Odd"}
```

In this lab, we represent an ID tree recursively as a tree of **`IdentificationTreeNode` objects**.

To begin, let's use an ID tree to classify a point! Classifying a point using an ID tree is straightforward. We recursively apply the current node's classifier to the data point, using the result to choose which branch to take to the child node. If a "leaf" node is ever encountered, that node gives the final classification of the point. Essentially, at each node, we make a recursive call on the node to follow the tree down the correct classifier.

Write a function `id_tree_classify_point` that takes in a point (represented as a dictionary) and an ID tree (represented as an `IdentificationTreeNode`) and uses the ID tree to classify the point (even if the point already has a classification defined), returning the final classification.

```
def id_tree_classify_point(point, id_tree):
```

This function can be written cleanly in 3-4 lines, either iteratively or recursively. If you're not sure how, check out the available methods in the **`IdentificationTreeNode` API**.

Part 1B: Splitting Data with a Classifier

In order to actually construct an ID tree, we'll need to work directly with *classifiers*, also known as *tests*. Recall that at each node of an ID tree (unless the training data at the node is *homogeneous*, i.e., the data at the node have the same final classification), we pick the best available classifier to split the data up. This same logic is repeated at every level of the tree, with each node taking as training data the data points sorted into that category by the node above it.

We represent each classifier using a **`Classifier` object**.

Implement `split_on_classifier`, which should take in a list of points and a particular `Classifier`, returning a dictionary mapping each possible feature value for that classifier (for example, obtained from calling `classifier.classify(point)`) to a list of the points that have that value:

```
def split_on_classifier(data, classifier):
```

For example, `split_on_classifier(angel_data, feature_test("Shape"))` should return a dictionary mapping "Human" to a list of four points and "Animal" to a list of two points.

Part 1C: Calculating Disorder

One of the first steps in constructing an ID tree is to calculate disorder. You may have learned about this idea in information theory as entropy. We'll start by coding the information-disorder equations to calculate the disorder of a branch or test.

However, before we start, we will review some nomenclature.

branch

A set of data representing the points that, upon applying a particular classifier (e.g. "Eats garlic") to a set of many points, all result in the same feature result. For example, suppose our data set is `[p1, p2, p3, p4, p5, p6]`, where each of `p1, ..., p6` are Python dictionaries representing data points. Then, for example, applying the "Eats garlic" classifier to the data could yield two branches, one for all of the points with `"Eats garlic": "Yes"` and one for all of the points with `"Eats garlic": "No"`.

test

Also known as a **decision stump** or **feature test**, and sometimes referred to loosely as a **classifier**. Encompasses all of the *branches* for a particular classifier. In the example for branches above, the two branches ("Yes" and "No") together constitute the decision stump for the "Eats garlic" classifier, often referred to concisely as the *test*.

disorder

A measure of how homogeneous (or not) a set of data points in a branch is. A branch whose points all have the same final classification is *homogeneous*, and has a disorder of 0. A branch with many different values has a very high disorder. The formula we use for disorder originates in information theory, and is described in detail on page 429 of **Winston's textbook (https://canvas.mit.edu/courses/4358/files/776479/download?wrap=1)** **(https://canvas.mit.edu/courses/4358/files/776479/download?wrap=1)**:

$$\text{Disorder}(\text{branch } b) = \sum_{\text{classification } c} -\frac{n_{bc}}{n_b} \log_2 \frac{n_{bc}}{n_b}$$

In this equation, n_b is the number of data points in the branch, and n_{bc} is the number of data points in the branch with the given classification. For example, a branch that has two points with classification `"Vampire"` and two points with classification `"Not Vampire"` has a disorder of 1.

As an example, suppose we have a data set representing different types of balls:

```
ball1 = {"size": "big", "color": "brown", "type": "basketball"}
ball2 = {"size": "big", "color": "white", "type": "soccer"}
ball3 = {"size": "small", "color": "white", "type": "lacrosse"}
ball4 = {"size": "small", "color": "blue", "type": "lacrosse"}
ball5 = {"size": "small", "color": "yellow", "type": "tennis"}
ball_data = [ball1, ball2, ball3, ball4, ball5]
```

Then, applying the classifier for "size" yields two branches:

```
branch1 = [ball1, ball2]
branch2 = [ball3, ball4, ball5]
```

where `branch1` represents balls that are big and `branch2` represents balls that are small. The disorder of `branch1` is 1, because the classifications in this branch are `["basketball", "soccer"]`. However, the disorder of `branch2` is roughly 0.9, because the classifications in this branch are `["lacrosse", "lacrosse", "tennis"]`.

However, applying the classifier for "color" instead yields

```
branch1 = [ball1]
branch2 = [ball2, ball3]
branch3 = [ball4]
branch4 = [ball5]
```

Here, the disorders of `branch1`, `branch3`, and `branch4` are all 0, and the disorder of `branch2` is 1.

Now, complete the function `branch_disorder`. This function should take in a list of data points *in the current branch* as well as the classifier for determining the final classification of the points, and return the disorder of the branch, as a number:

```
def branch_disorder(data, target_classifier):
```

For example, calling `branch_disorder([ball3, ball4, ball5], ball_type_classifier)` should yield approximately `0.918`.

Next, use your `branch_disorder` function to help compute the disorder of an entire test (a decision stump). Recall that the disorder of a test is the weighted average of the disorders of the constituent branches, where the weights are determined by the fraction of points in each branch.

`average_test_disorder` should take in a list of points (the data), a Classifier (the feature test), and the `target_classifier` for determining the final classification of the points; it then computes and return the disorder of the entire test, as a number:

```
def average_test_disorder(data, test_classifier, target_classifier):
```

For example, calling `average_test_disorder(ball_data, size_classifier, ball_type_classifier)` should yield approximately `0.959`, because the "small" branch has weight `3/5` and a disorder of about `0.918`, while the "big" branch has a weight of `2/5` and a disorder of `1`.

Part 1D: Constructing an ID Tree

Using the disorder functions you defined above, implement a function to select the best classifier. The function takes in four arguments:

- `data`: data, as a list of point dictionaries
- `possible_classifiers`: a list of classifiers, as `Classifier` objects
- `target_classifier` the target classifier (a `Classifier` object) that the ID tree uses to ultimately classifies points: e.g. a classifier representing the test "Vampire?"

The function should return the classifier that has the lowest disorder.

Edge cases:

- If multiple classifiers are tied for lowest disorder, break ties by preferring the classifier that occurs earlier in the list.
- If the classifier with lowest disorder wouldn't separate the data at all (that is, the classifier has only one branch), **raise the exception** `NoGoodClassifiersError` instead of returning a classifier.

```
def find_best_classifier(data, possible_classifiers, target_classifier):
```

Now, it is time to build an ID tree!

The function `construct_greedy_id_tree` takes in four arguments:

- `data`: a list of points dictionaries
- `possible_classifiers`: a list of `Classifier` objects that you can use in constructing your tree
- `target_classifier`: the `Classifier` that the ID tree uses to ultimately classify points (e.g. a Classifier representing "Vampire?")
- `id_tree_node` (optional): an incomplete tree, represented as an `IdentificationTreeNode`. If the incomplete tree is provided, finish it; if it is not provided, create a new tree using the constructor: `id_tree_node = IdentificationTreeNode(target_classifier)`

Then, add classifiers and classifications to the tree until either perfect classification has been achieved, or there are no good classifiers left.

We recommend implementing this function recursively, which can be done cleanly in about 15 lines. Your base case should occur when you encounter a leaf node, in which case you should set the node's classification. The recursive step occurs when your classifier splits the data into groups, in which case you should set the classifier, expand the node (hint: `set_classifier_and_expand`), and recursively construct the tree for each new branch.

As a general outline:

1. Once the current id_tree_node is defined, perform one of three actions, depending on the input node's data and available classifiers:
   - If the node is homogeneous, then it should be a leaf node, so add the classification to the node.
   - If the node is not homogeneous and the data can be divided further, add the best classifier to the node.

- If the node is not homogeneous but there are no good classifiers left (i.e. no classifiers with more than one branch), leave the node's classification unassigned (which defaults to `None`).

2. If you added a classifier to the node, use recursion to complete each subtree.
3. Return the original input node.

```
def construct_greedy_id_tree(data, possible_classifiers, target_classifier, id_tree_node=None):
```

Congrats, you're done constructing ID trees!

We've provided some datasets from past quizzes, so you can now use your ID tree builder to solve problems! For example, if you run

```
print(construct_greedy_id_tree(tree_data, tree_classifiers, feature_test("tree_type")))
```

it should compute and print the solution to the tree-identification problem from 2014 Q2.

You can also try:

```
print(construct_greedy_id_tree(angel_data, angel_classifiers, feature_test("Classification")))
# from 2012 Q2
print(construct_greedy_id_tree(numeric_data, numeric_classifiers, feature_test("class")))  # fro
m 2013 Q2
```

You can also change the target_classifier attribute to, for example, use `tree_type` to predict what type of bark_texture a tree has:

```
print(construct_greedy_id_tree(tree_data, tree_classifiers_reverse, feature_test("bark_textur
e")))  # build an ID tree to predict bark_texture
```

Optional: Construct an ID tree with real medical data

Constructing ID trees for small datasets (such as the ones from quiz problems) is straightforward and generally produces nice, simple trees. But what happens if you construct an ID tree on a real-life dataset with hundreds of points and over a dozen features? In a real medical situation, such as in an emergency room, doctors need a way to quickly determine which patients need immediate attention. If a patient comes in complaining of chest pain, doctors may use an ID tree to ask the patient a series of medical questions to determine the likelihood that the patient has heart disease, or to distinguish between patients with heart disease, pneumonia, and broken ribs (all of which may cause chest pain, but require different treatments and different levels of urgency).

We've provided a dataset **cleveland_medical_data.txt (https://canvas.mit.edu/courses/4358/files/776476/download?wrap=1)**  **(https://canvas.mit.edu/courses/4358/files/776476/download?wrap=1)** of anonymized medical data

collected by the Cleveland Clinic Foundation. The dataset contains information on 303 patients who were being checked for heart disease. For each patient, it includes 13 features:

- Age: int (in years)
- Sex: M or F
- Chest pain type: typical angina, atypical angina, non-anginal pain, or asymptomatic (Note that "angina" means "chest pain".)
- Resting blood pressure: int (in mm Hg)
- Cholesterol level: int (in mg/dl)
- Is fasting blood sugar < 120 mg/dl: Yes or No
- Resting EKG type: normal, wave abnormality, or ventricular hypertrophy
- Maximum heart rate: int
- Does exercise cause chest pain?: Yes or No
- ST depression induced by exercise: int
- Slope type: up, flat, or down
- # of vessels colored: float or '?' (This is the number of major vessels (0-3) colored by flourosopy.)
- Thal type: normal, fixed defect, reversible defect, or unknown

(If you don't understand all the medical terminology, don't worry -- you can still use the data!)

Each patient also has a known classification, represented both on a binary scale and on a discrete scale:

- Heart disease presence: healthy or diseased
- Heart disease level: int 0-4, where 0 means healthy (no presence of heart disease), and 1-4 indicate different levels of heart disease, with 4 being the most severe

In parse.py, we've parsed the dataset for you and stored it in the variable `heart_training_data`. `heart_training_data` is a list of 303 dictionaries, where each dictionary represents a patient and looks something like this:

```
{'name': 'patient280', 'Cholesterol level': 282, 'ST depression induced by exercise': 1, 'Age': 65, 'Resting EKG type': 'ventricular hypertrophy', 'Chest pain type': 'typical angina', 'Does exercise cause chest pain?': 'No', 'Sex': 'M', 'Thal type': 'normal', '# of vessels colored': 1.0, 'Is fasting blood sugar < 120 mg/dl': 'Yes', 'Resting blood pressure': 138, 'Maximum heart rate': 174, 'Heart disease level': 1, 'Slope type': 'flat', 'Heart disease presence': 'diseased'}
```

We've also defined a list of 345 Classifiers, most of which are numeric threshold classifiers. As recommended in lecture, we included a threshold classifier between every pair of neighboring possible values for a feature. For example, if one patient has a cholesterol level of 150 and another

has 156, but there are no values in between, we'll add a Classifier for "Cholesterol level > 153". All of the classifiers are stored in the variable `heart_classifiers`.

Finally, we've defined two possible target classifiers:

- `heart_target_classifier_binary`, for the binary "Heart disease presence", and
- `heart_target_classifier_discrete`, for the discrete "Heart disease level".

To construct an ID tree with this dataset, all you need to do is add `from parse import *` to your lab5.py file, then call `construct_greedy_id_tree` with the heart data, heart classifiers, and target classifier of your choice.

What do you notice? Does this seem like the simplest possible tree? If not, why not? What could we change to make it simpler?

Try using the other target classifier (binary or discrete). Is the tree simpler or more complicated? Why?

Try using `tree.print_with_data(data)` to print the training data at the leaf nodes. What do you notice?

To diagnose a new patient, first create a dictionary of their attributes, for example:

```
test_patient = {\
    'Age': 20, #int
    'Sex': 'F', #M or F
    'Chest pain type': 'asymptomatic', #typical angina, atypical angina, non-anginal pain, or as
ymptomatic
    'Resting blood pressure': 100, #int
    'Cholesterol level': 120, #int
    'Is fasting blood sugar < 120 mg/dl': 'Yes', #Yes or No
    'Resting EKG type': 'normal', #normal, wave abnormality, or ventricular hypertrophy
    'Maximum heart rate': 150, #int
    'Does exercise cause chest pain?': 'No', #Yes or No
    'ST depression induced by exercise': 0, #int
    'Slope type': 'flat', #up, flat, or down
    '# of vessels colored': 0, #float or '?'
    'Thal type': 'normal', #normal, fixed defect, reversible defect, or unknown
}
```

Then use your `id_tree_classify_point` function or `tree.print_with_data([test_patient])` to classify the patient.

If you get an error such as `KeyError: '?'` when using `id_tree_classify_point`, it's not a bug in your code; it just means that the ID tree is unable to classify the point. Either define a different patient, or use `tree.print_with_data` to see why the point is unclassifiable.

Important Legal Disclaimer

> Please do not use this to perform real medical diagnosis. As the questions below will show, there are numerous factors that may cause your ID tree to mis-classify patients.

Try creating and classifying a few patients. (You can copy/paste the `test_patient` code into your `lab5.py` file to define your own patients.) Are the results consistent with your expectations?

What happens if a female patient has `'Thal type': 'unknown'`? What happens if a male patient has `'Thal type': 'unknown'`? If you're surprised by the results, examine your tree and data to figure out what caused the results. (You can use `tree.print_with_data` to print your tree with all the training points.) What could we change to improve the classification accuracy of patients with unknown Thal type?

In the discrete classification tree, what happens if a patient has `'Thal type': 'normal'`, `'Chest pain type': 'asymptomatic'`, and `'# of vessels colored': '?'`? Why does this happen? Why might it cause a problem for classifying real patients? What can we do to fix this problem?

(For answers to the questions in this section, look at the bottom of `parse.py`.)

Part 1E: Multiple Choice

Questions 1-3: Identification (of) Trees

These questions refer to the data from 2014 Quiz 2, Problem 1, Part A, which is stored in the variables `tree_data` (the data) and `tree_classifiers` (a list of Classifiers) in `data.py`.

Question 1: Which of the four classifiers (a.k.a. feature tests) has the lowest disorder? Fill in ANSWER_1 with one of the four classifiers as a string: 'has_leaves', 'leaf_shape', 'orange_foliage', or 'bark_texture'.

Question 2: Which of the four classifiers has the *second* lowest disorder? Fill in ANSWER_2 with one of the four classifiers as a string.

Question 3: If we start constructing the greedy, disorder-minimizing ID tree (with final classifier `tree_type`), we'll start with the classifier from Question 1. Our one-classifier tree has exactly one non-homogeneous branch. For that branch, which of the four classifiers has the lowest disorder? Fill in ANSWER_3 with one of the four classifiers as a string.

Questions 4-7: XOR

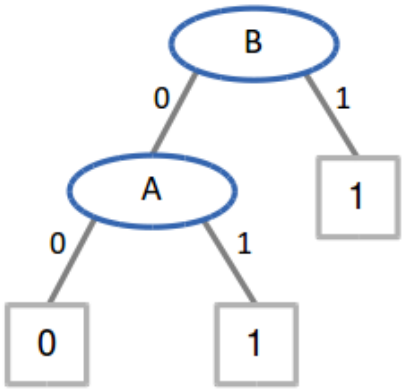
For questions 4-7, we encourage you to think about these logically before plugging the data into your code. These questions will help develop your intuition for ID tree construction.

Each training point in the dataset below has three binary features (A, B, C) and a binary classification (0 or 1). Note that the classification happens to be the boolean function XOR(A,B), while feature C is

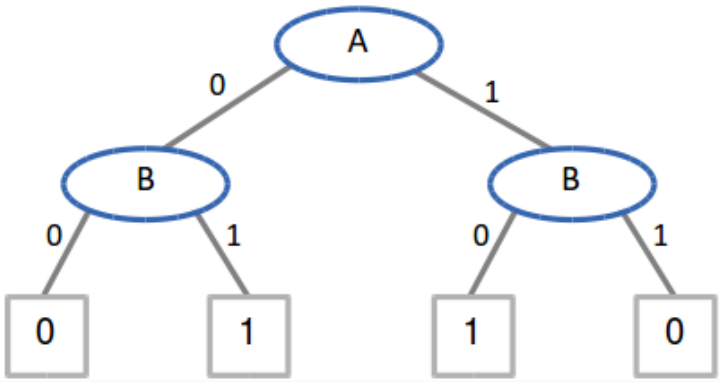
just random noise. (The dataset is available in `data.py` as `binary_data`, and the three classifiers are stored in the variable `binary_classifiers`.)

Classification	A	B	C
0	0	0	0
0	0	0	1
1	0	1	0
1	0	1	1
1	1	0	1
1	1	0	1
0	1	1	0

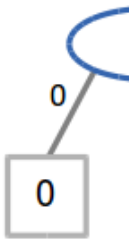
Consider the three identification trees below. They are defined in the lab code as `binary_tree_n`, for `n` from 1 to 3. To answer the questions below, you may use your code or work out the answers by hand.



binary_tree_1



binary_tree_2



Question 4: Which of the trees correctly classify all of the training data? Fill in ANSWER_4 with a list of numbers (e.g. `[1,2]` if trees 1 and 2 correctly classify the training data but tree 3 does not, or `[]` if no tree correctly classifies the training data).

Question 5: Which of the trees could be created by the greedy, disorder-minimizing ID tree algorithm? Fill in ANSWER_5 with a list of numbers.

Question 6: For *any* binary dataset with features A, B, and C, which trees correctly compute the function Classification=XOR(A,B)? Fill in ANSWER_6 with a list of numbers.

Question 7: Based on the principle of Occam's Razor, which tree (among the ones that correctly classify the data) is likely to give the most accurate results when classifying unknown test points? Fill in ANSWER_7 with a single number, as an int.

Questions 8-9: General properties of greedy ID trees

These questions are about ID trees in general and do not refer to any specific trees, data, or classifiers.

Question 8: When constructing an ID tree, in the first round we pick a classifier for the top node, in the second round we pick classifiers for the children nodes, and so on. With this in mind, when constructing a greedy, disorder-minimizing ID tree, is the classifier with the *second* lowest disorder in the first round always the same as the classifier with the *lowest* disorder in the second round? (Hint: Consider your answers to Questions 1-3.) Answer 'Yes' or 'No' in ANSWER_8.

Question 9: Will a greedy, disorder-minimizing tree always be the simplest possible tree to correctly classify the data? (Hint: Consider your answers to Questions 4-7.) Answer 'Yes' or 'No' in ANSWER_9.

For explanations of the answers to these questions, search for "#ANSWER_n" in tests.py, for the appropriate value of n.

This ends Part 1 for Lab 6. If you're ready to move on to Part 2, you can find it here: **Part 2 (kNN) (https://canvas.mit.edu/courses/4358/pages/lab-6-part-2-k-nearest-neighbors)** .

API

In `lab5.py`, we have defined for you...

- `log2`: a function representing the logarithm to base 2, taking in a single number and returning $\log_2$ of the number.
- `INF`: a constant representing positive infinity, as in previous labs

In addition, the file `api.py` defines the `Classifier` and `IdentificationTreeNode` classes, as well as some helper functions for creating `Classifier` objects, all described below.

Classifier

`Classifier` objects are used for constructing and manipulating ID trees.

A `Classifier` has one attribute and one method:

`name`

The name of the classifier.

`classify(point)`

A function which takes in a point and returns its classification. A classification can be a string (e.g. `"Vampire"`), a boolean value (`True` or `False`), or an int.

In our ID trees, a point is represented as a dict mapping feature names to their values. For example:

```
point = {"X": 1, "Y": 2, "Zombie": True}
```

You can create new `Classifier` objects using the following provided methods:

- `feature_test(key)`: Takes in the name of a feature (such as `"X"` or `"Accent"` or `"Height"`). Returns a new `Classifier` that classifies based on the value of that feature.
- `threshold_test(key, threshold)`: Takes in the name of a numeric feature and a threshold value. Returns a new binary `Classifier` that performs the test `point[key] > threshold` and returns `"Yes"` or `"No"` for each point.

For example:

```
feature_test_X = feature_test("X")
feature_test_X_threshold = threshold_test("X", 6)
```

Or you can create your own `Classifier` using the `Classifier` constructor. For example:

```
feature_test_vector_length = Classifier("vector_length", lambda pt: (pt.get("X")**2 + pt.get
("Y")**2)**0.5)
```

IdentificationTreeNode

In this lab, an ID tree is represented *recursively* as a tree of `IdentificationTreeNode` objects. In particular, an `IdentificationTreeNode` object fully represents an entire ID tree rooted at that node.

For example, suppose we have an `IdentificationTreeNode` called `id_tree_node` Then, `id_tree_node`'s children are themselves `IdentificationTreeNode` objects, each fully describing the sub-trees of `id_tree_node`. However, if `id_tree_node` has no children, then `id_tree_node` is a leaf, meaning that it represents a homogeneous (by classification) set of data points. Furthermore, any datum at this node is definitively classified by that leaf's classification.

As such, in a completed ID tree, each node is either

- a leaf node with a *classification* such as `"Vampire"` or `"Not Vampire"` in the vampires example; `or`
- a non-leaf node with a *classifier* (such as `"Accent"` in the vampires example), with branches (one per classifier result, e.g. `"Heavy"`, `"Odd"`, and `"None"`) leading to child nodes.

An `IdentificationTreeNode` has the following attributes and methods:

`target_classifier`

The single **`Classifier`** by which the tree classifies points (e.g. a `Classifier` representing `"Vampire?"` for the vampires example, or `"Classification"` for the angel data).

`get_parent_branch_name()`

Returns the name of the branch leading to this node, or `None` if this is a root node.

`is_leaf()`

Returns `True` if the node is a leaf (has a classification), otherwise `False`.

`set_node_classification(classification)`

Sets this node's classification, thus defining this node as a leaf node. Modifies and returns this object. May print warnings if the node already has branches defined.

`get_node_classification()`

Returns this node's classification, if it has one. Otherwise, returns `None`.

`set_classifier_and_expand(classifier, features)`

Takes in a **`Classifier`** object and a list of features (e.g. `["Heavy","Odd","None"]`), then uses the `classifier` and list of features to set the current node's classifier and add branches below the current node, instantiating a new child node for each branch. Modifies and returns this object. May print warnings if the specified `classifier` is inadvisable. (For your convenience, `features` can also be a dictionary whose keys are feature names.)

`get_classifier()`

Returns the **classifier** associated with this node, if it has one. Otherwise, returns `None`.

`apply_classifier(point)`

Applies this node's classifier to a given data point by following the appropriate branch of the tree, then returns the *child* node. If this node is a leaf node (and thus doesn't have a classifier), raises a `ClassifierError`.

`get_branches()`

Returns a dictionary mapping this node's branch names to its child nodes, e.g. `{'Heavy': <IdentificationTreeNode object>, 'Odd': <IdentificationTreeNode object>, 'None': <IdentificationTreeNode object>}`

`copy()`

Returns a (deep) copy of the node

`print_with_data(data)`

Given a list of points, automatically classifies each point and prints a graphical representation of the ID tree with each point at the appropriate node.

The constructor for instantiating a new `IdentificationTreeNode` requires 1 or 2 arguments:

- `target_classifier`: The **`Classifier`** by which the overall tree should classify points
- `parent_branch_name` (optional): The name of the branch leading into this node, if the node is not the root node

For example:

```
vampire_tree_root_node = IdentificationTreeNode(feature_test("Vampire?"))
another_node = IdentificationTreeNode(feature_test("Vampire?"), "Odd")
```

Appendix: How to handle exceptions

(using NoGoodClassifiersError as an example)

Python command	
`raise NoGoodClassifiersError("message")`	Stops execution by raising, or throwing, a NoGoodCla
`try:` `    #code here`	Defines the beginning of a block of code that might ra
`except NoGoodClassifiersError:` `    #code here`	Follows a `try` block. Defines what to do if a NoGoodC