Programming Assignment 3: Spam Classifier

Background

Seemingly everyone is talking about Machine Learning and Artificial Intelligence these days. Artificial Intelligence (AI) is a subfield of Computer Science concerned with using computers to automate rational thinking. AI is one of the oldest research topics in computer science, and interest in AI has driven improvements in computer technology since at least the 1950s. Machine Learning (ML) is a subfield of AI that uses trends from previous examples to predict things about unseen data using statistical methods. ML algorithms are not magic -- they simply guess the most likely outcome based on many, many previous examples. This means that any ML algorithm's predictions are only as good as the data it was built upon, which can easily be biased in some way, or just wrong. As computer scientists, it is important to be able to recognize and advocate for appropriate uses of these models, regardless of how miraculous they may seem to the public.

Terminology

There are several machine learning terms used throughout the specification for this assignment that we would like to formally define before you begin. It might even be worth having this slide open in another tab while reading the assignment to make sure you fully understand the terms being given to you.

- **Model:** The actual program that makes probabilistic classifications on provided inputs.
- **Training:** Models are "trained" on previously gathered datasets to make future predictions.
- **Label**: How data is classified after being run through the model. In our tree, leaf nodes will house classification labels.
- **Split**: Some way of differentiating one classification from another for different inputs. In our tree, intermediary nodes will house splits. Each split defines a feature and a threshold to determine which direction to travel:
    - **Feature**: Important aspects/characteristics of our dataset that we use in classification that corresponds to a numeric value. Typically, the hardest part of a machine learning algorithm is determining how to take input data and "featurize" it into something a computer can understand
        - Ex: turning a sentence or image into a series of numbers.
    - **Threshold**: The numeric value we're comparing a feature against at any split within our classifier. In our tree, if the current input is less than the threshold we should go left. If it's greater than or equal to, we should go right.

Specification

Learning Objectives

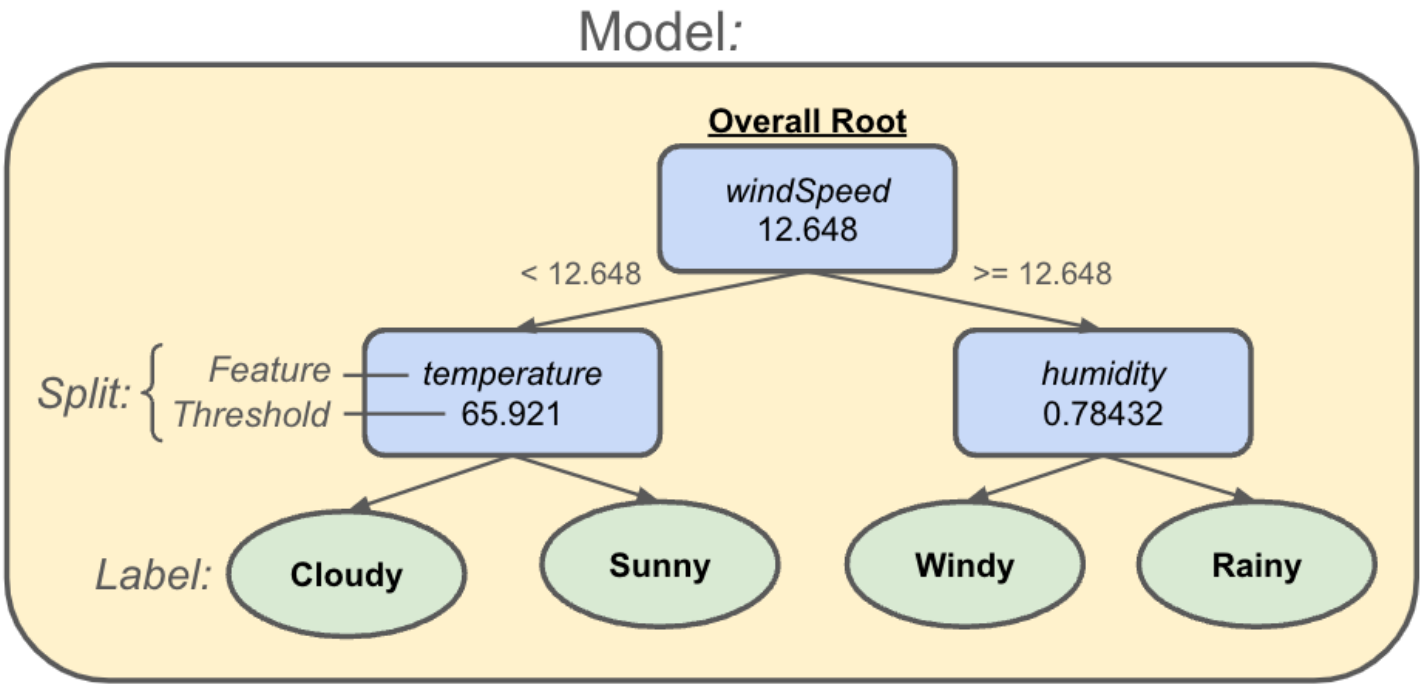
By completing this assignment, students will demonstrate their ability to:

- Implement a well-designed Java class that extends an abstract class to meet a given specification.
- Understand and correctly use various Machine Learning terminology
- Define data structures to represent compound and complex data
- Write a functionally correct Java class to represent a binary tree.
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, implementation, and performance.
- Produce clear and effective documentation to improve comprehension and maintainability of programs, methods, and classes.

Assignment

This assignment involves a lot of Machine Learning (ML) terminology that is further defined in the Background slide. For clarity, these terms are underlined within this specification

Your goal for this assignment is to implement a classification tree, a simplistic machine learning model that given some input data will predict some label for it. Below is a visual example of what a classification tree might look like for some weather data. It also includes relevant labels for each of the vocab terms defined on the last slide.



As seen above, in our classification tree the **leaf nodes represent our predictive labels** (Cloudy, Sunny, Windy, or Rainy) while the **intermediary nodes represent a split** on some feature of our data (windSpeed, temperature, or humidity). To reach a classification for some input, you start at the root of the tree and determine whether the corresponding feature falls to the left or right of the current node's threshold (determined by < or >=) and travel in the corresponding direction. Repeating this process will eventually lead you to a classification for your input.

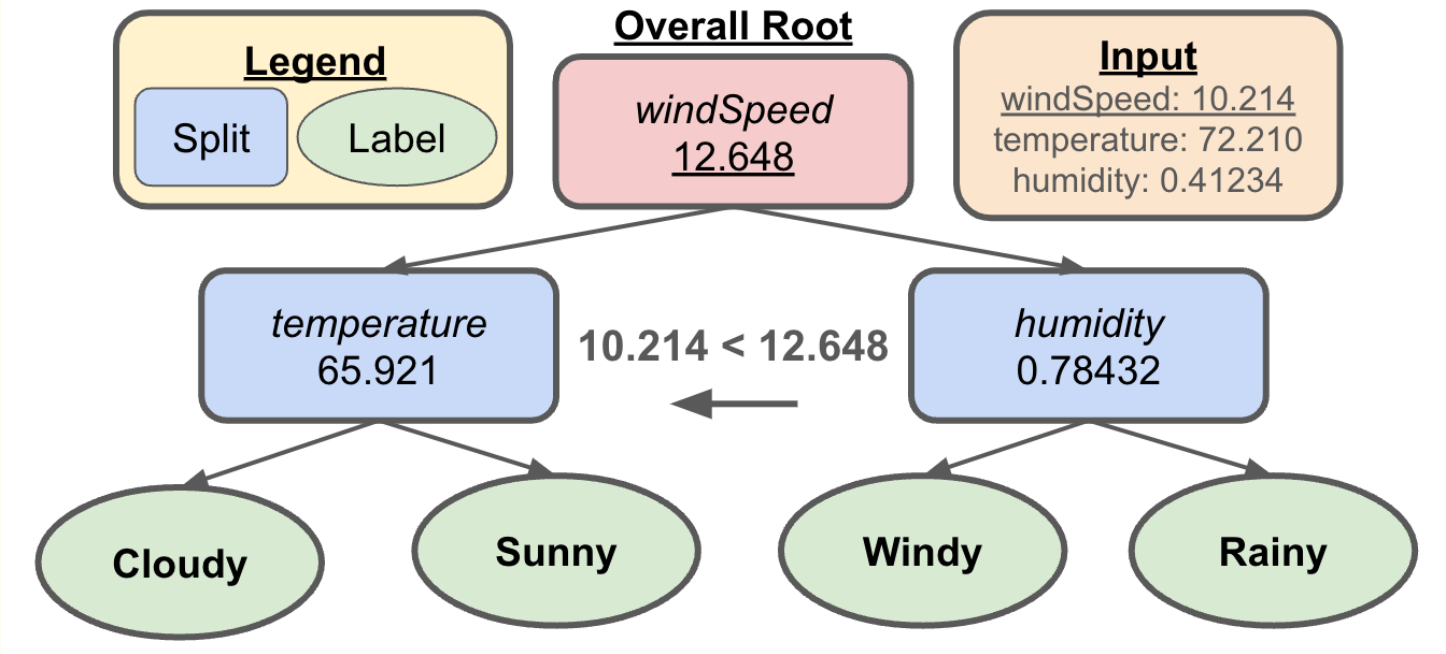
Below we'll trace through a sample input with our example weather model.

▼ Expand

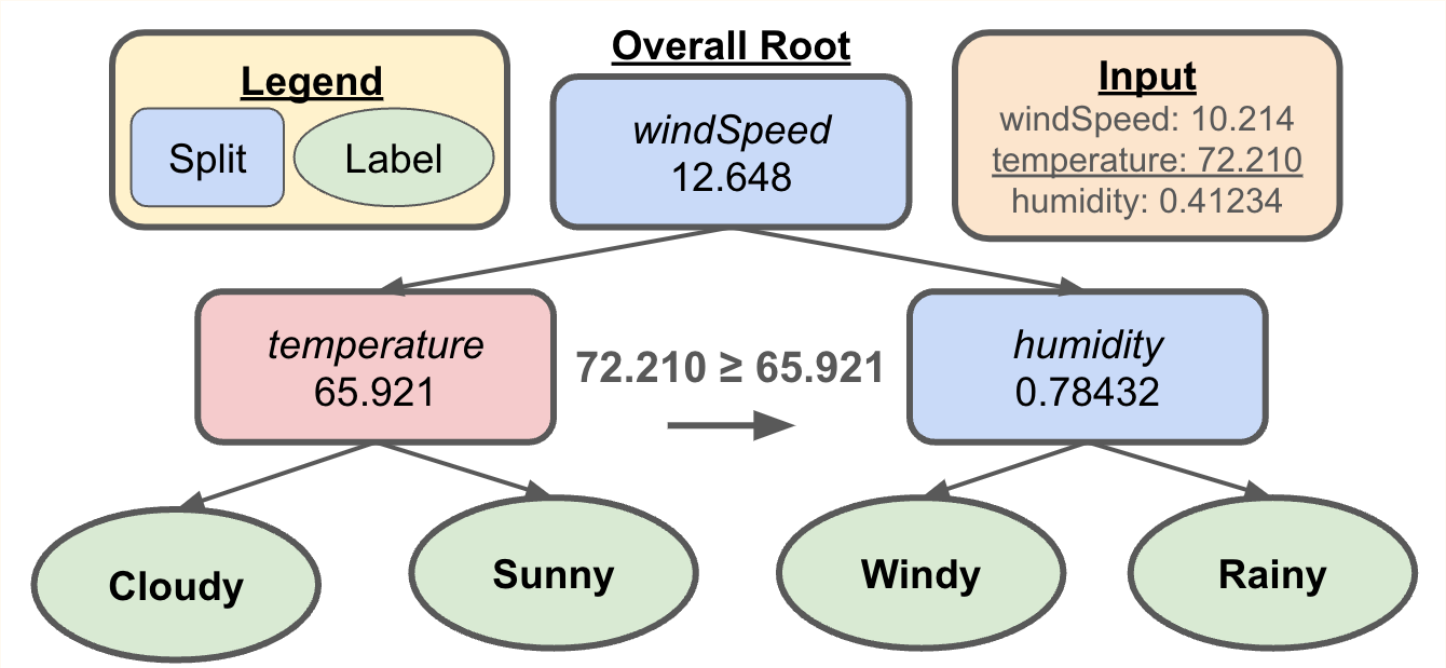
We'll begin at the root node with a `Classifiable` object (**"Input"**) containing the following features and their values:

```
windSpeed (10.214), temperature (72.210), humidity (0.41234)
```

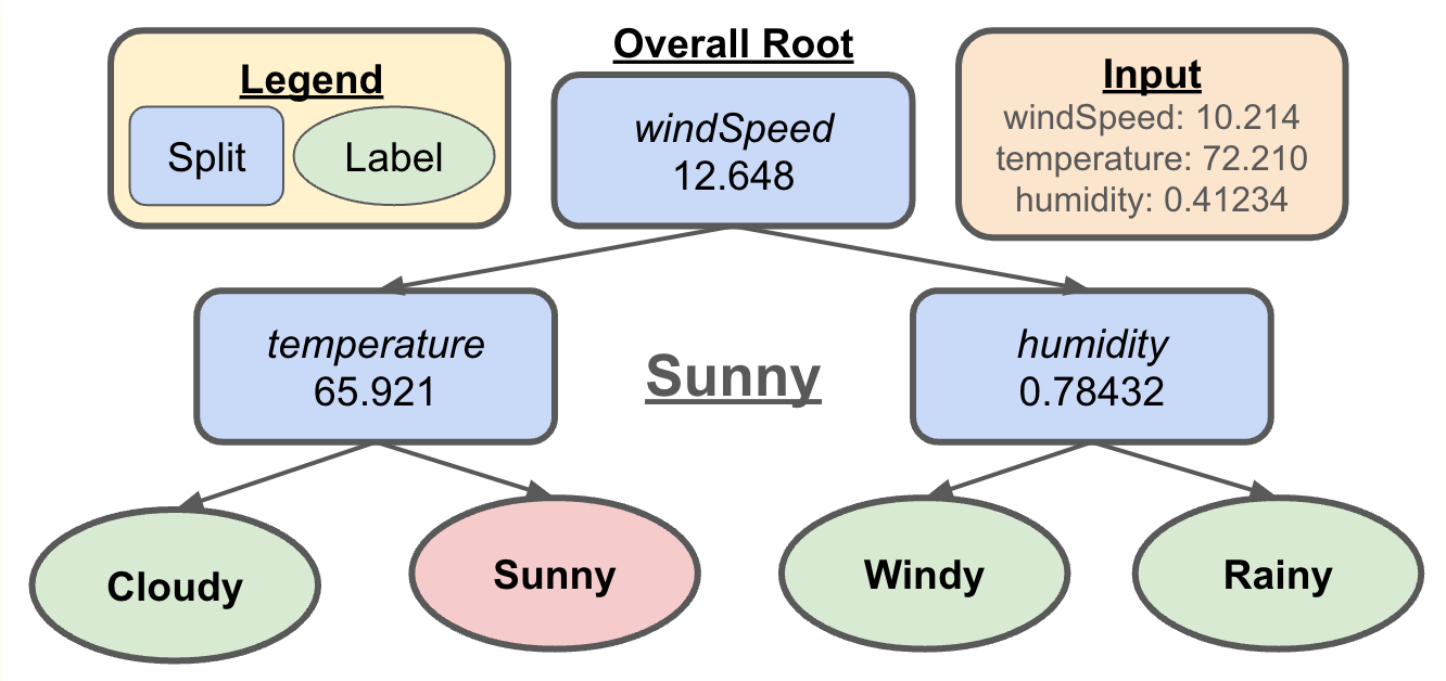
1. Since the windSpeed feature of the input is < the threshold (12.648) we'll travel left to the temperature node



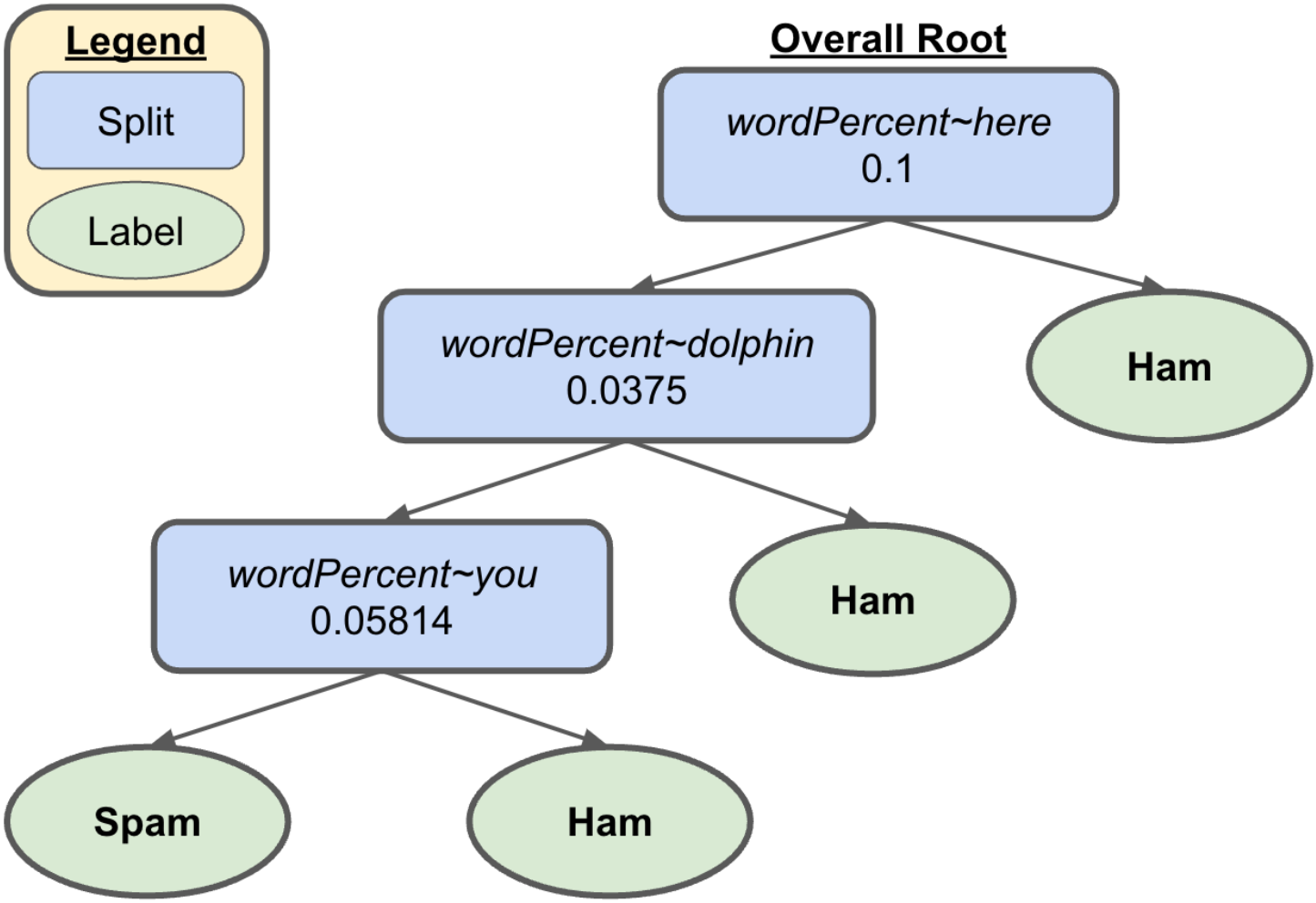
2. Since the temperature feature of the input is >= the threshold (65.921) we'll travel right to the Sunny node



3. We have reached a leaf node and therefore can predict that input corresponds to a sunny day (the resulting label)



Another example of how a classification tree might be used is for spam email classification. Below is an alternative example of what a potential classification tree might look like in this case.



Similar to the above, you'll notice that the leaf nodes of this tree represent labels ("Spam" or "Ham" – a funny way of writing not spam) while the intermediary values represent a split on some feature of our data (wordPercent). Notice that the features in this example are slightly different from the weather one above. Specifically, wordPercent is the only feature within this model; however, we also need to track the specific word we're comparing the percentage of. This is accomplished by appending the word preceded by some arbitrary "splitter" character (in this case '~') that separates the two.

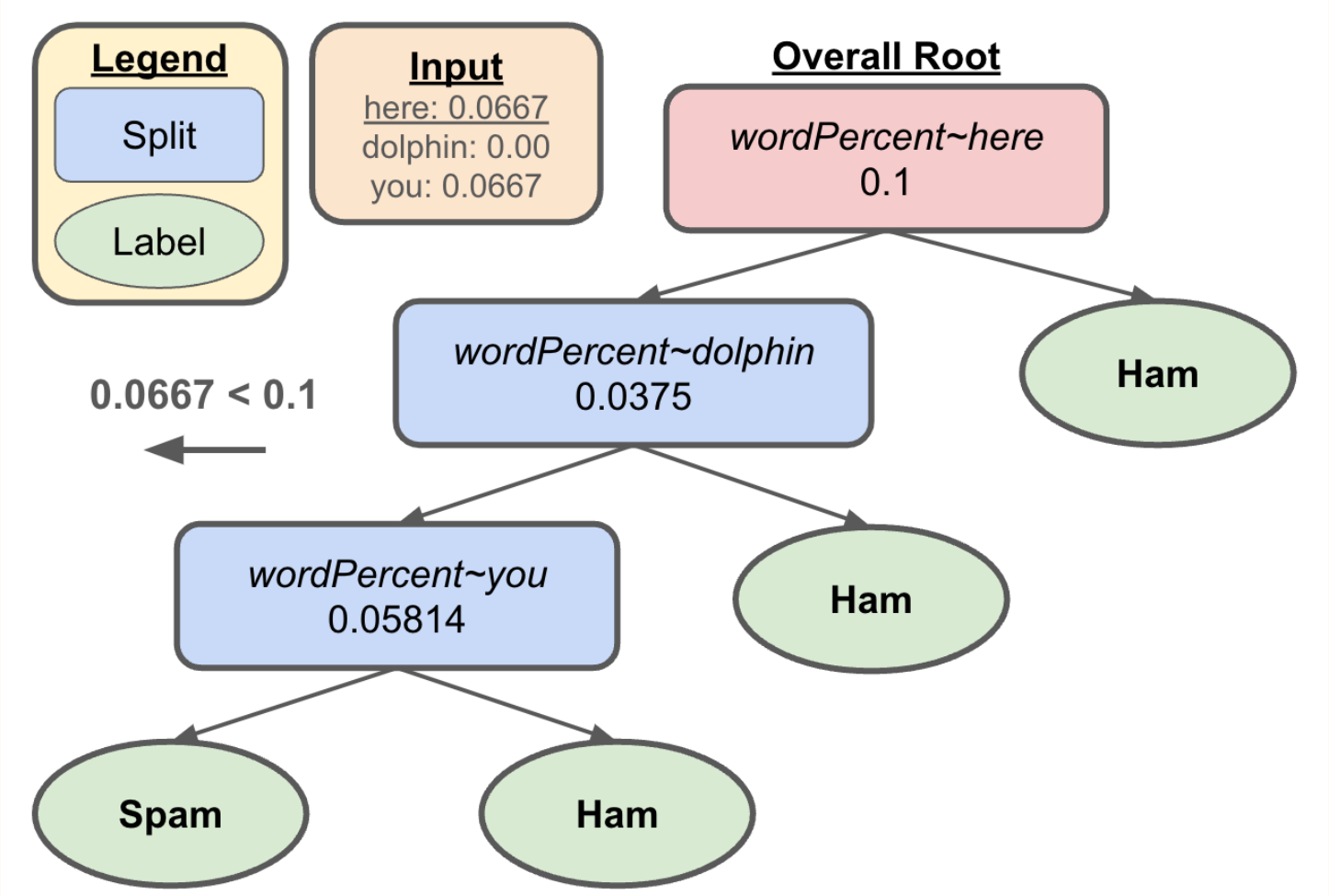
To solidify this idea, we'll trace through an input much like the weather example above.

▼ Expand

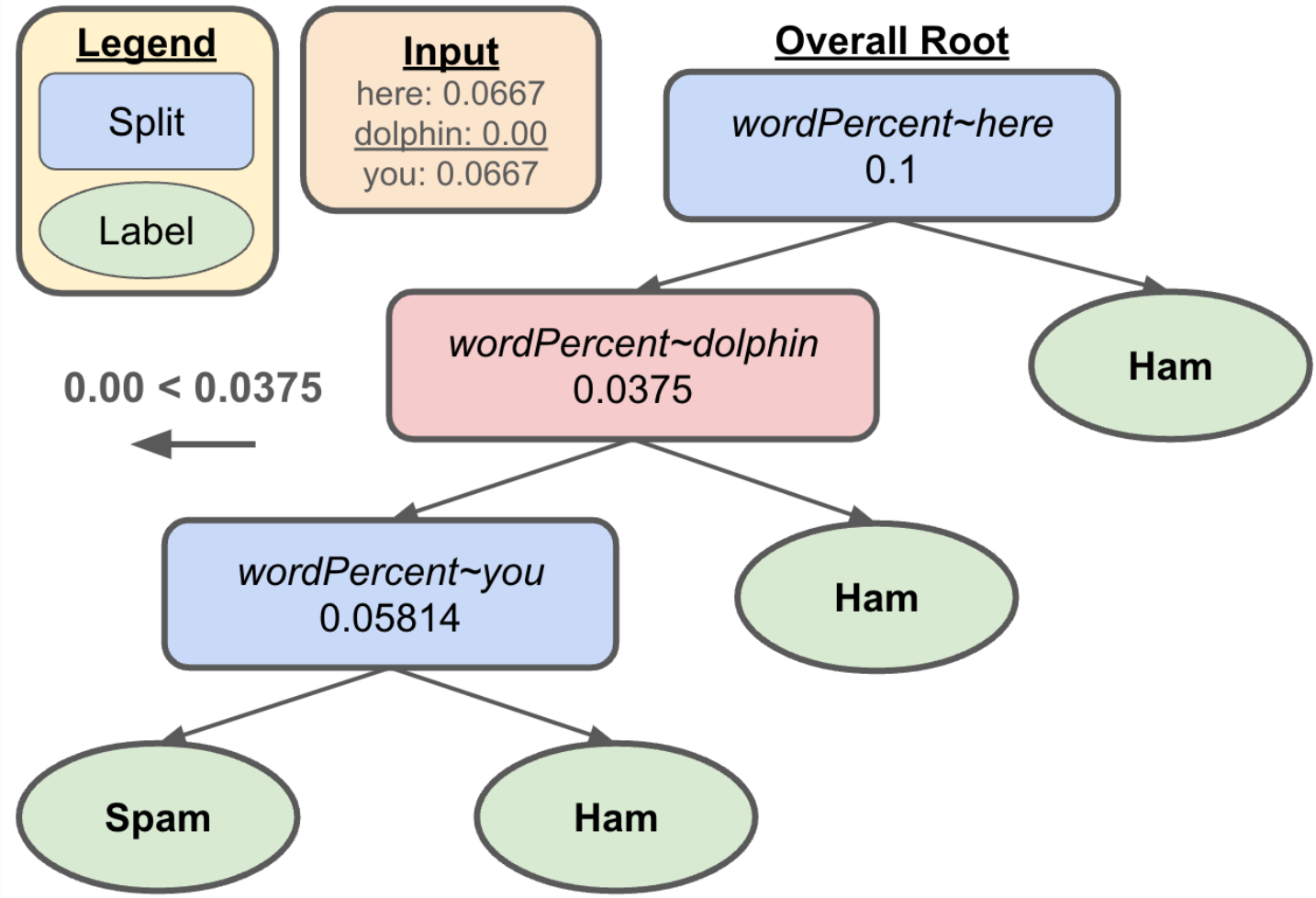
We'll begin at the root node with the following input:

```
        content
input:  hello, i am here at your office but the door is locked. are you there?
```

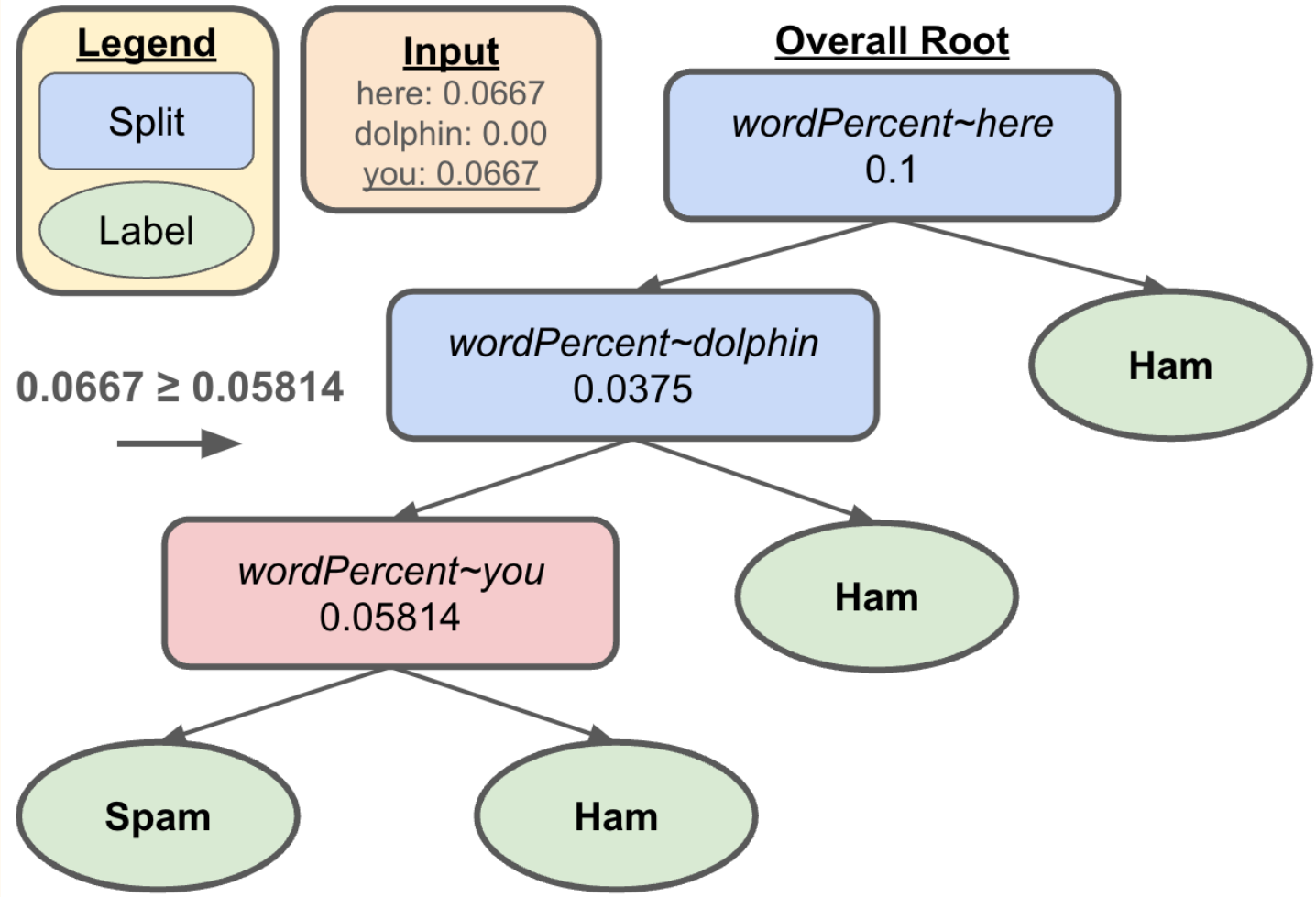
1. Since 'here' consists of 6.67% of the input email, which is < the threshold (10.00%) we'll travel left to the wordPercent~dolphin node.



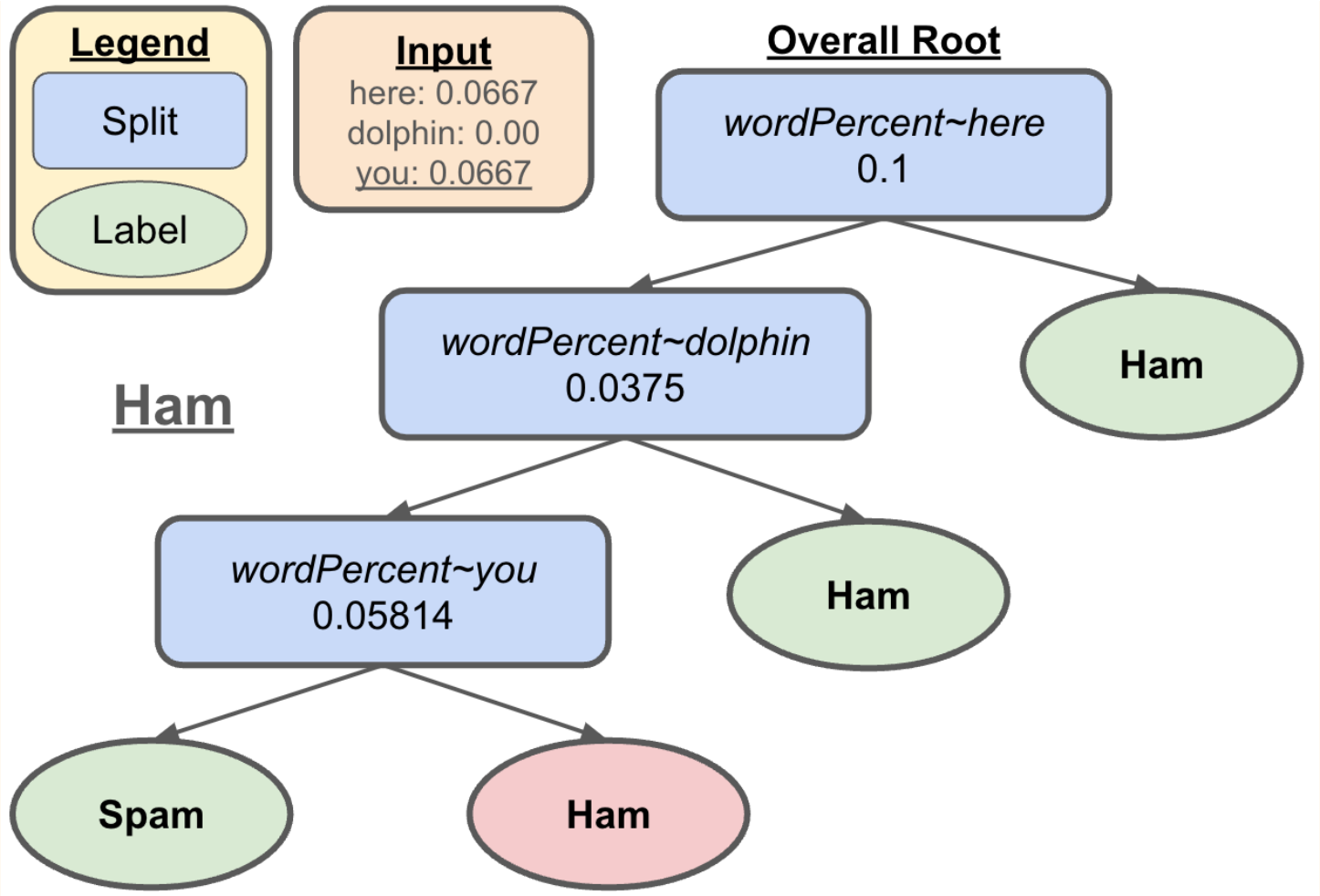
2. Since 'dolphin' consists of 0.00% of the input email, which is < the threshold (3.75%) we'll travel left to the wordPercent~you.



3. Since 'you' consists of 6.67% of the input email, which is >= the threshold (5.814%) we'll travel right to the Ham node.



4. We have reached a leaf node and therefore can predict that input corresponds to a Ham email (the resulting label).



This is what you'll be implementing in this assignment! Specifically, you'll be creating a classification tree that's able to predict given some email whether it's "Spam" or "Ham"!

System Structure

Below we describe three provided classes that will aid you in your implementation of `ClassificationTree.java`. Make sure to understand the purpose of these classes and read through the provided documentation.

Classifiable.java

(Given) All data that gets classified via the classifier (e.g. `Email.java`) implements this interface. It defines three methods:

▼ Expand

```
public double get(String feature);
```

- Returns the corresponding value for the given feature.
    - Although there are classification trees where it would make sense to return something

else (imagine a color feature within a real estate dataset), since our implementation is only dealing with thresholds this must return a double.

```
public Set<String> getFeatures();
```

- Returns a set of all features for a given dataset. This is useful in determining **whether or not this type of data point can be classified by a specific `Classifier`.**

```
public Split partition(Classifiable other);
```

- Returns a partition (`Split`) between this data point and `other`.
  - How this is computed is up to the implementer (and is a large part of the complexity of our model).
  - Note that there is no difference between calling `one.partition(two)` and `two.partition(one)`. Both will return `Splits` with the same feature and threshold.
  - **Example:** In the `Email` class, calling `partition()` could return a new Split containing `Feature: wordPercent~later Threshold: 0.125`

A simple example of all the above implementations can be seen in the provided `Email` class.

Split.java

(Given) To help implement your node class, we have provided the `Split` class: a wrapper class that you should use to store both a feature and threshold for any **intermediary** (non-leaf) nodes within your tree. Below are some methods that will likely be useful in your implementation:

▼ Expand

```
public Split(String feature, double threshold)
```

- Constructs a new `Split` with the given feature and threshold

```
public String getFeature()
```

- Returns the feature name without any specific component tied to it.
  - In the case of our email example, it would return "wordPercentage" without the specific word tied to it (instead of "wordPercentage~dolphin")

```
public boolean evaluate(Classifiable value)
```

- Evaluates the provided value `Classifiable` object on this split, returning true if it falls below (<) this split, and false if it falls above.
  - In other words, given some Classifiable data, return **whether you should travel left or right** from this point.

```
public String toString()
```

- Returns a String representation of the given Split in the following format:

```
Feature: <feature>
Threshold: <threshold>
```

Classifier.java

(Given) The class you're required to implement must extend the `Classifier` abstract class provided in the coding workspace. Below is a description of these methods and hints for useful methods within other classes.

▼ Expand

This is an abstract class that any model implementation must extend to prove it is capable of classifying some Classifiable data (see starter code) input. Below are the three abstract methods of `Classifier`:

```
public abstract boolean canClassify(Classifiable input);
```

- Given a piece of classifiable data, returns whether or not this tree is capable of classifying it.
  - You can imagine that it wouldn't make much sense to try and run an email input through our weather classifier above, which is why this method is useful! A tree is capable of classifying an input if all features within the tree (**see `Split.getFeature`**) are contained within the input's valid features (**see `Classifiable.getFeatures`**).

```
public abstract String classify(Classifiable input);
```

- Given a piece of classifiable data, return the appropriate label that this classifier predicts.
  - This method should model the steps taken in our weather example above: at every split point, evaluate (**see `Split.evaluate`**) our input data and determine if it's less than our threshold. If so, continue left; otherwise, continue right. Repeat this process until a leaf node is reached.
- If the input is unable to be classified by this classifier, this method should throw an `IllegalArgumentException`.

```
public abstract void save(PrintStream ps);
```

- Saves this current classifier to the given PrintStream
  - For our classification tree, **this format should be pre-order**. Every intermediary node will print two lines of data, one for feature preceded by "Feature:" and one for threshold preceded by "Threshold:" (**see `Split.toString`**). For leaf nodes, you should only print the label. **Examples of the format can be seen below and through the**

`trees` **directory in the start code.**

NOTE: This class also implements a `calculateAccuracy` method that returns the model's accuracy on all labels in a provided testing dataset. This is useful to see how well our model works, and what labels it is struggling with classifying correctly.

Required Operations

ClassificationTree.java

For this assignment, you're **only** required to implement `ClassificationTree.java` a class that extends `Classifier` but with the following additional constructors:

```
public ClassificationTree(Scanner sc)
```

- Load the classification tree from a file connected to the given Scanner. `sc` should not be null and the format of the input file should match that of the `save` method described within `Classifier`.
    - Importantly, in this method, you should only read data from the file using `nextLine` and convert it to the appropriate format using `Double.parseDouble`.

```
public ClassificationTree(List<Classifiable> data, List<String> results)
```

- Create a classification tree from the input data and corresponding labels.
    - **Note that you are building the tree up from scratch in this constructor**.
- The lists should be processed in parallel, where the label corresponding to `data.get(i)` can be found at `results.get(i)`. The general construction process should be accomplished via the algorithm described below. There will be two parts:
    - Traverse through the current classification tree until you reach a leaf node.
        - If the node's label matches the current label, do nothing (our model is accurate up to this point).
        - If the label is incorrect, create a split between the data used to create the original leaf node***** and our current input.
            - **HINT**: Use `Classifiable.partition` to generate this split
            - Insert a new intermediary node that uses this split to correctly classify the current data and the old data.
- This method should throw an `IllegalArgumentException` if the provided lists aren't the same size or the lists are empty.

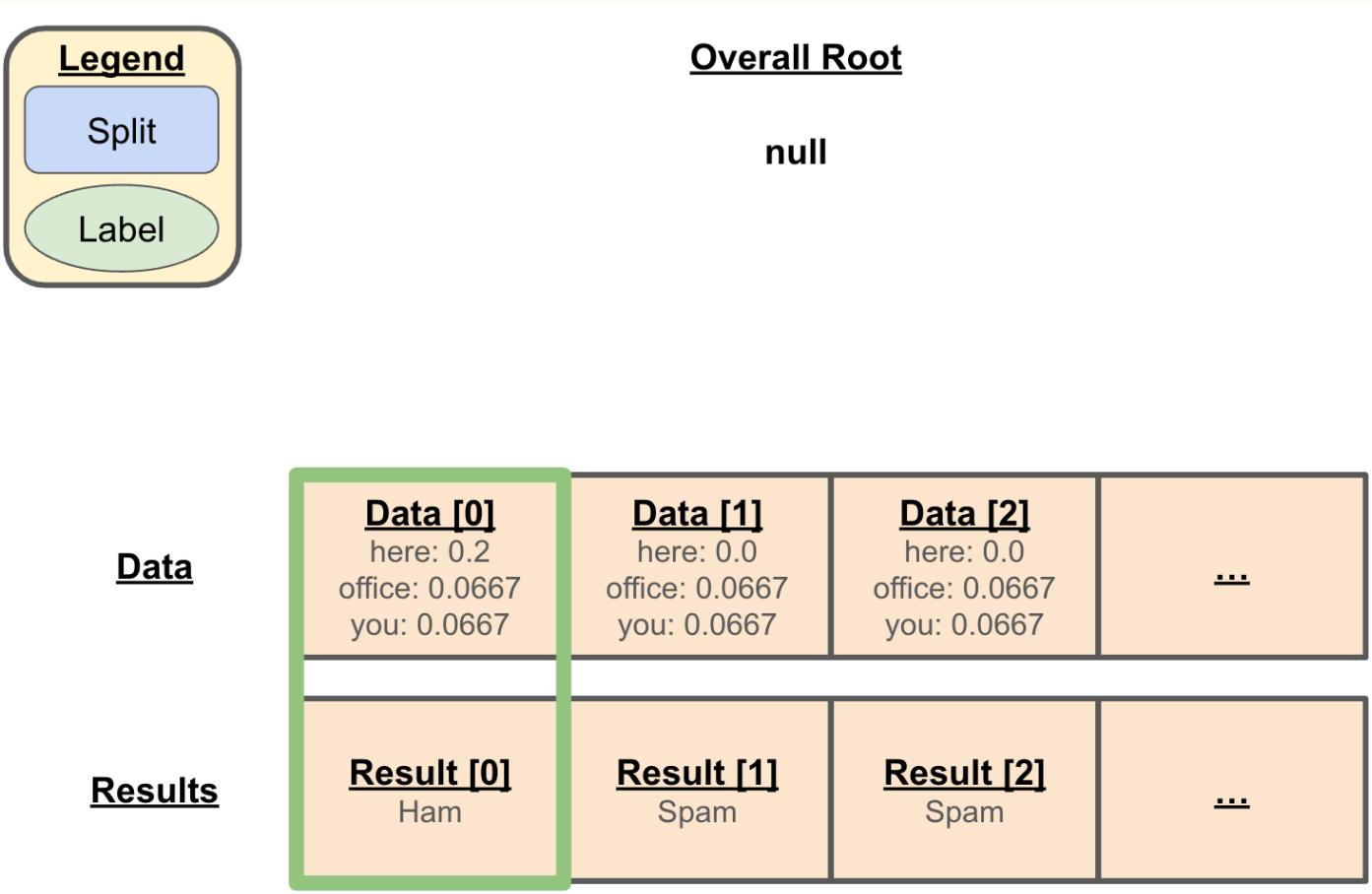
***** This algorithm requires you to keep track of both the label and the Classifiable datapoint first assigned to this label within every leaf node created in this constructor, as without the previous Classifiable datapoint we would be unable to create a new split! Ideally we'd like to keep track of all input data that falls under a specific leaf node such that when creating a new split, we can make sure it's valid for our entire training dataset. For

simplicity, only worry about the first datapoint used to create a label node.

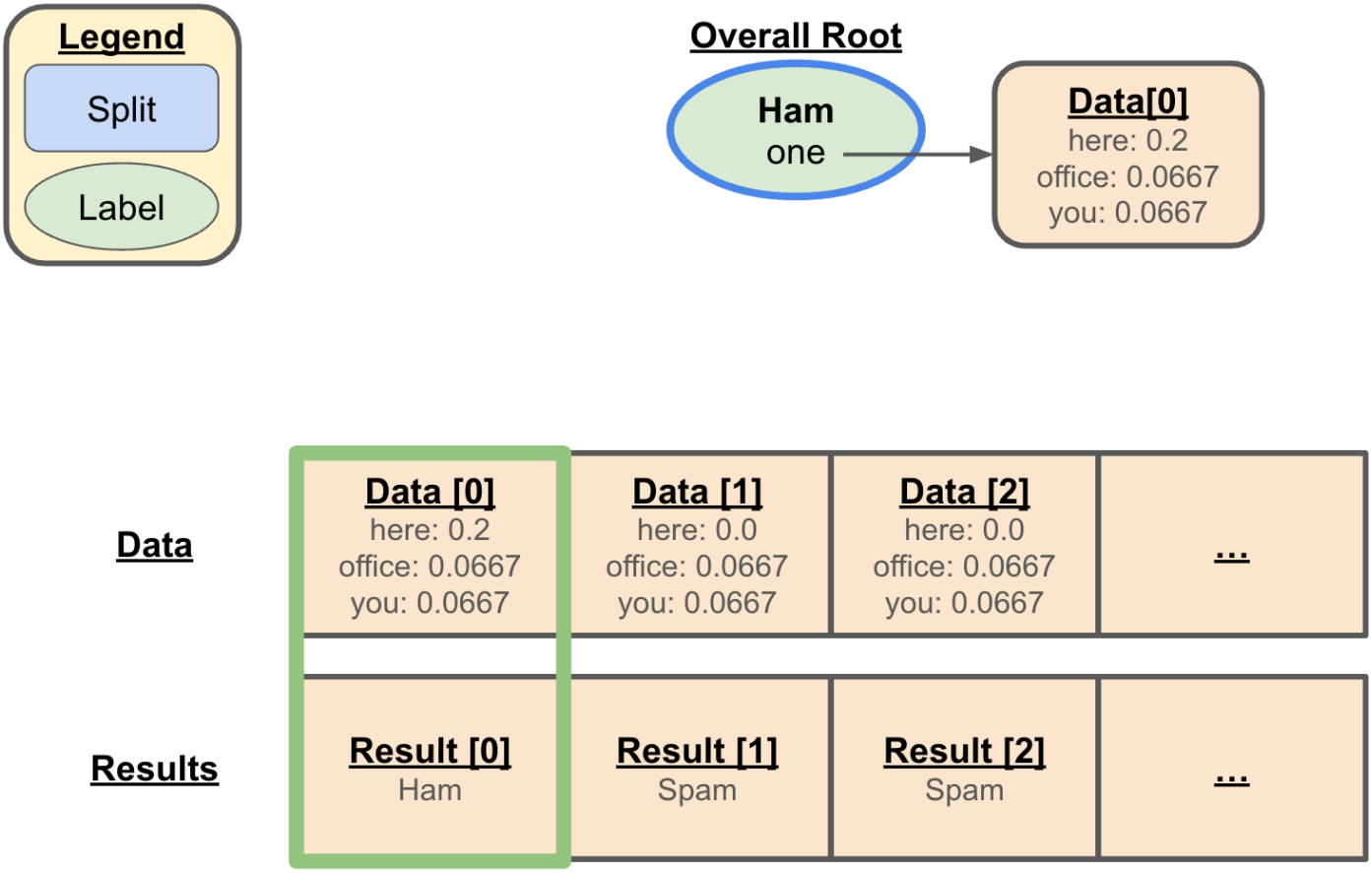
The algorithm above is further shown in the following diagrams:

▼ Expand

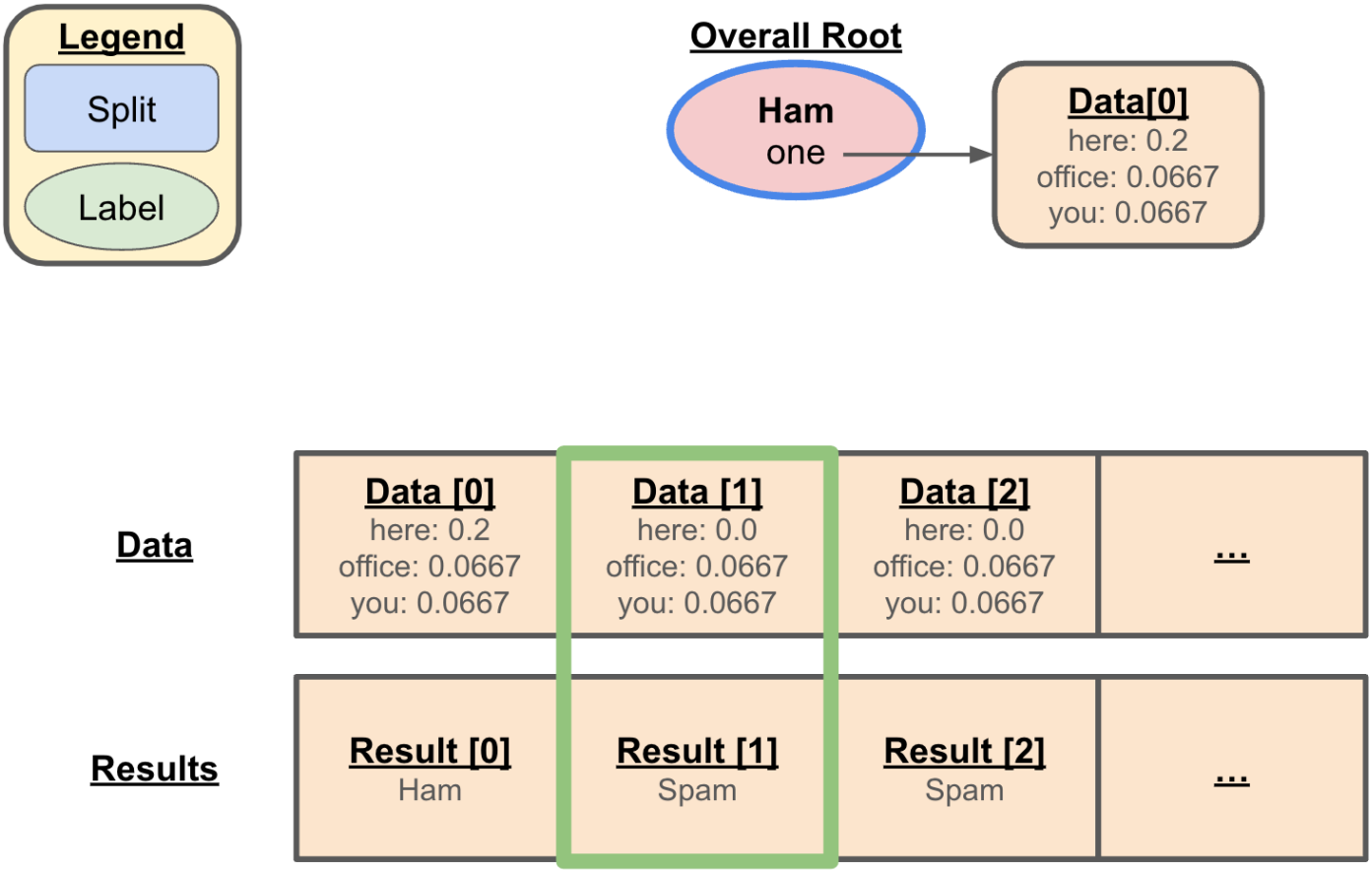
Before we begin our algorithm, we need to make sure there is a non-empty tree we can traverse. We start with an empty tree and process the first input:



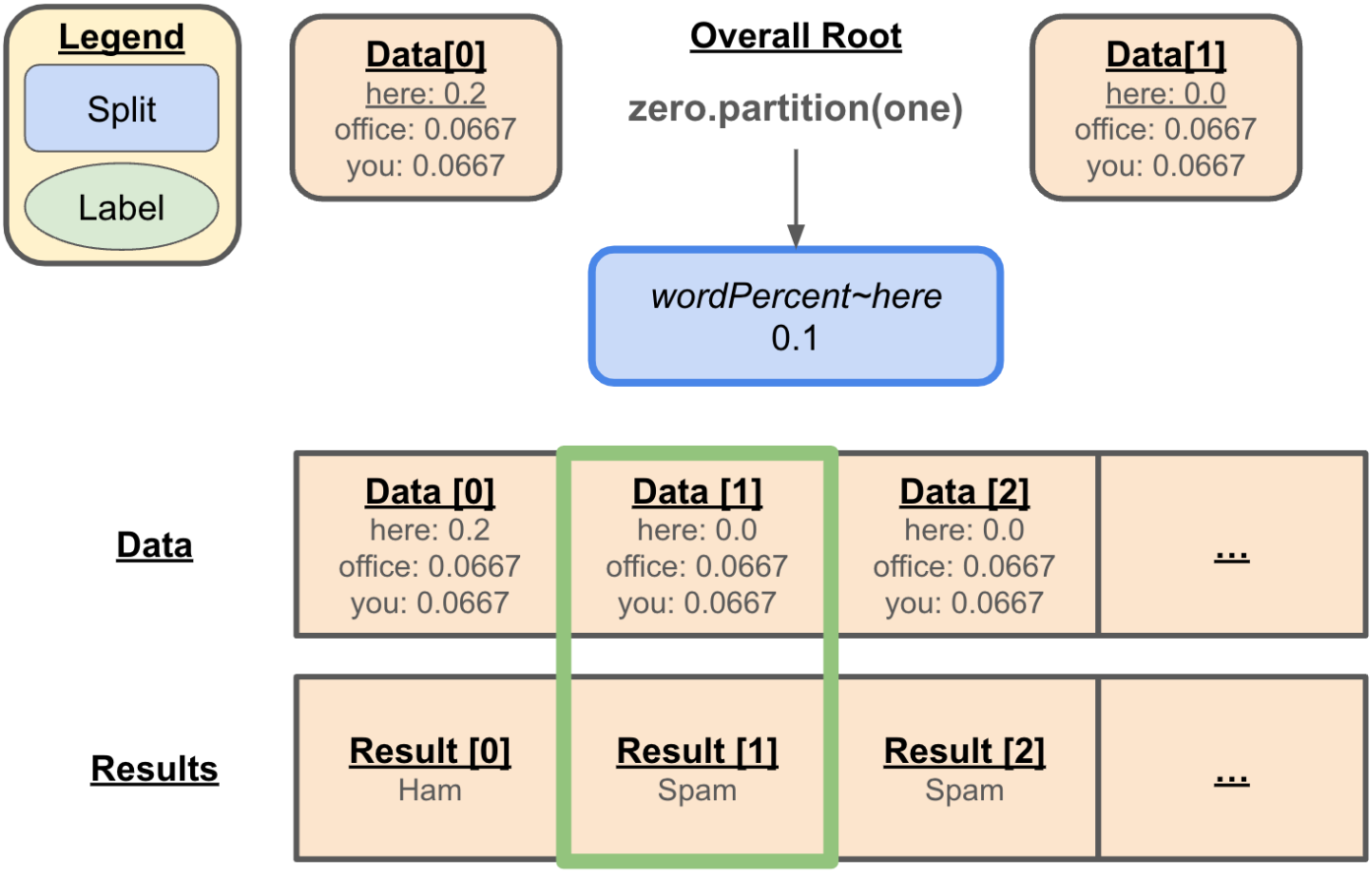
At the very beginning of our constructor, we should fill our empty tree with a single node containing the appropriate label of the first data point:



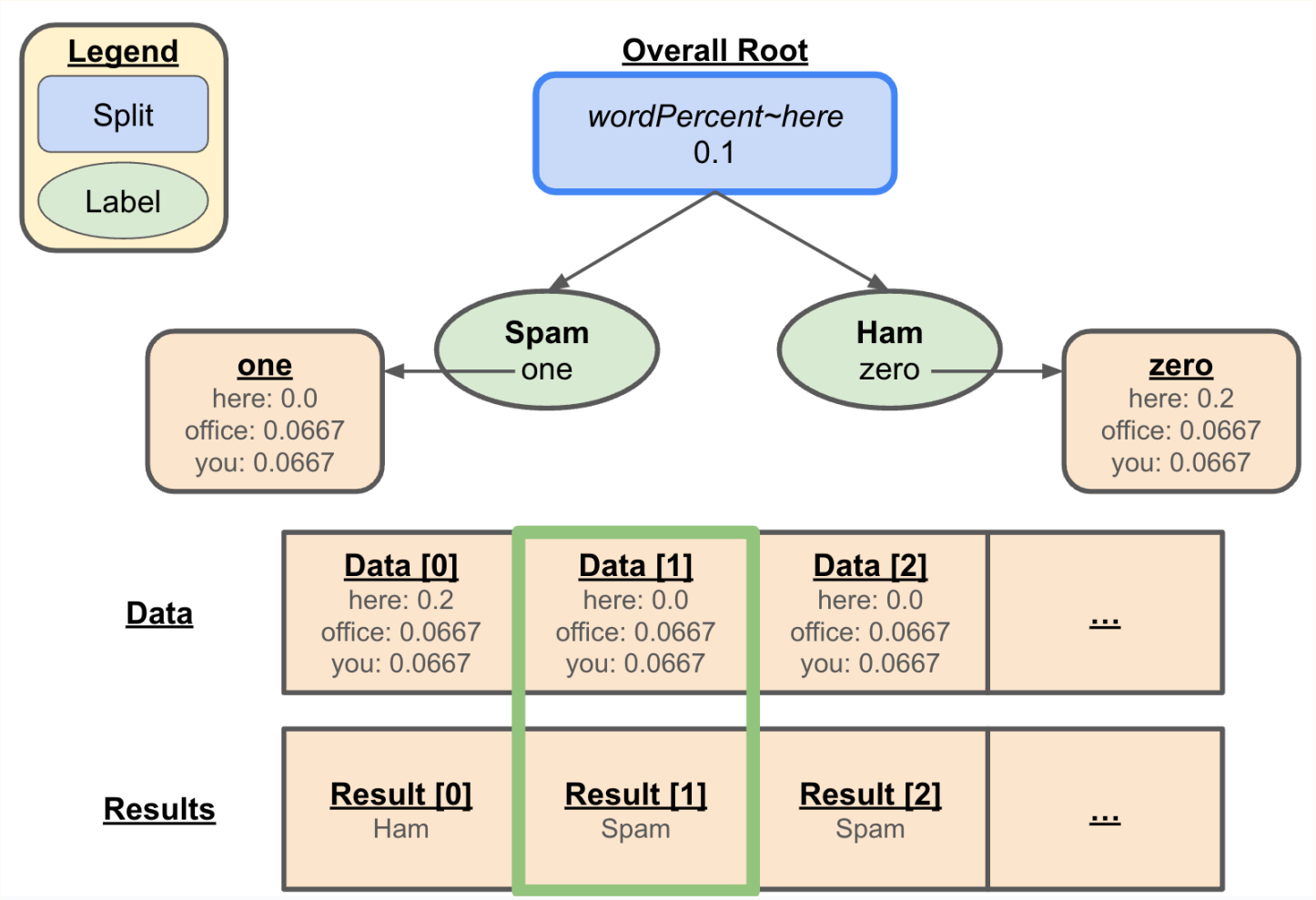
Note that this node also stores a reference to the data used to create it. This will be useful in the next step. Once we've processed the first data point, we move on to the second. Now, we can follow the algorithm and traverse through the existing tree until reaching a leaf node (which just so happens to be the only node in our tree):



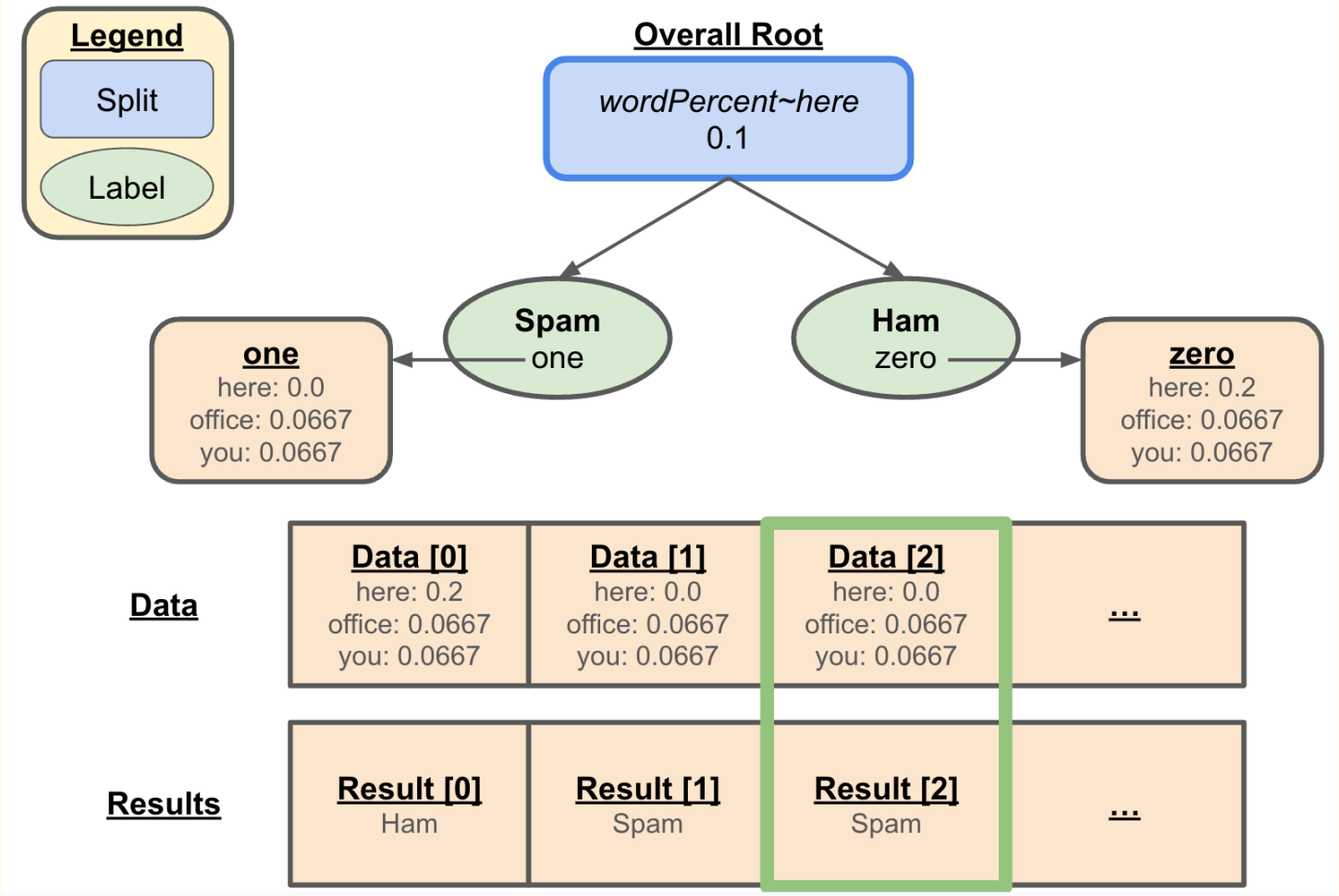
Then we see if the resulting label is correct. Our expected result is "Spam", but the one predicted by our model is "Ham". This is incorrect, so we need to create a new split via the `Classifiable.partition()` method:



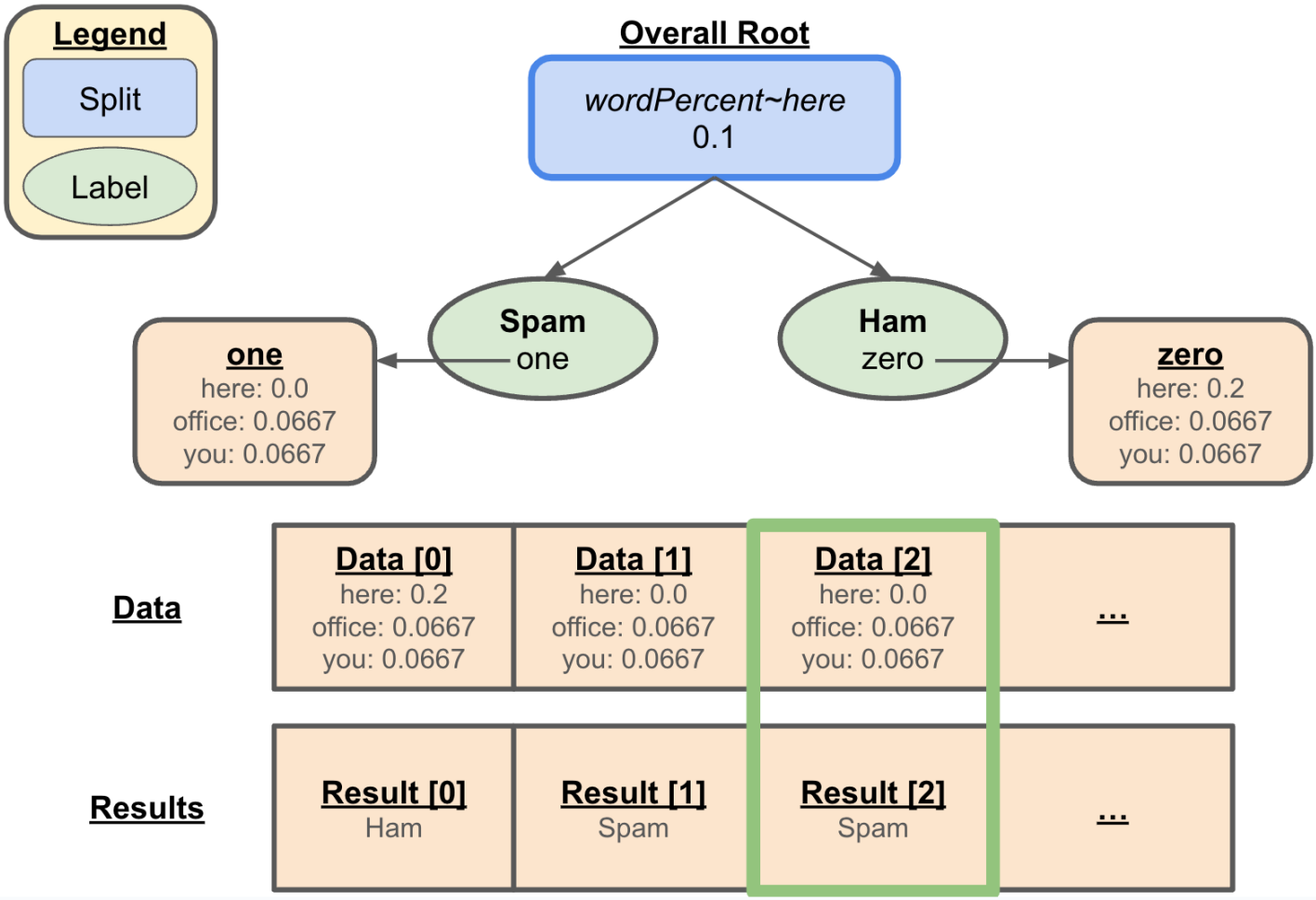
This will return a new `Split` which we can then store within a new intermediary node that will allow us to correctly distinguish `one` vs. `two`. All that's left to do is organize the label nodes appropriately as seen below:



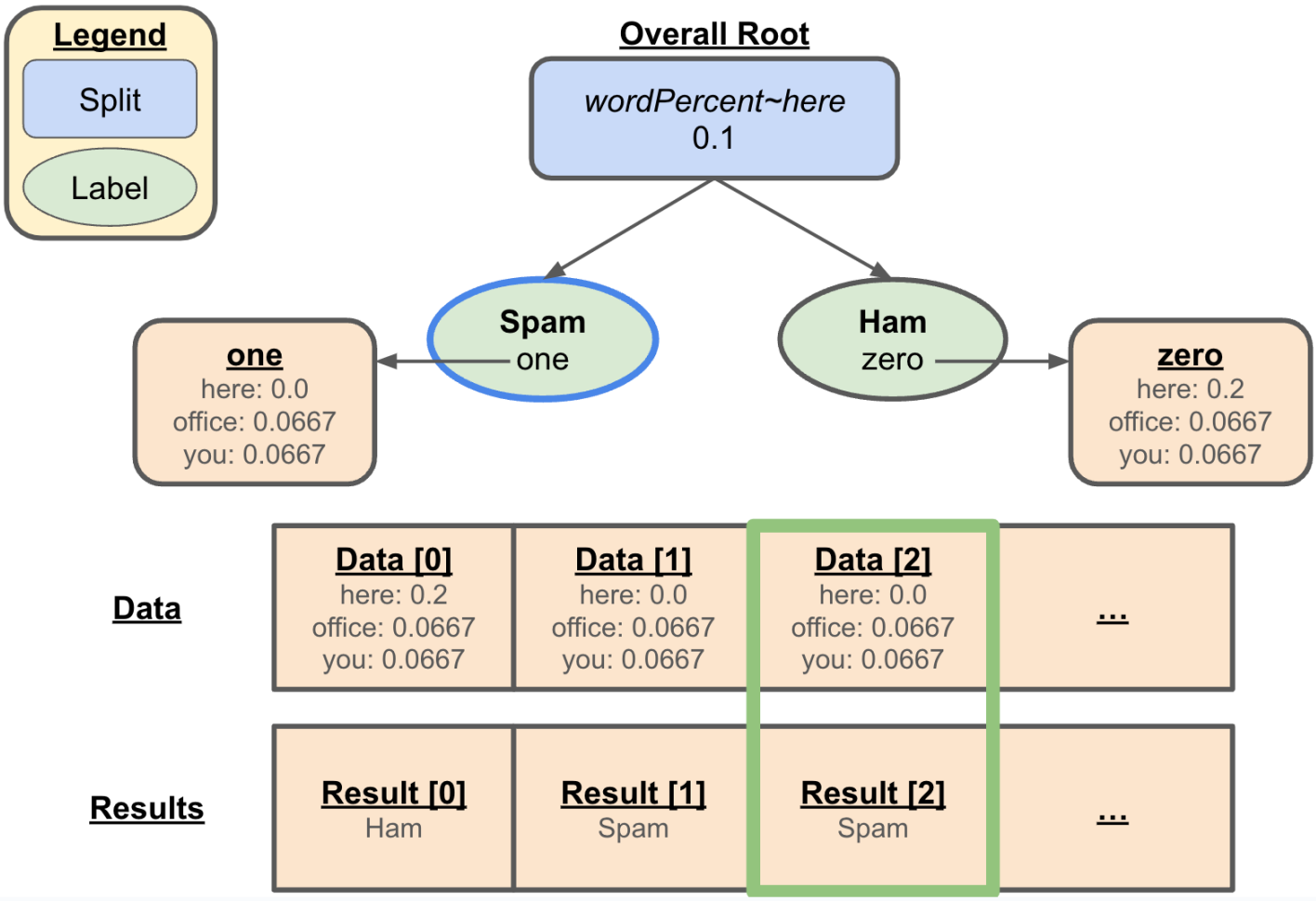
Furthermore, we can imagine undertaking this process with a third data point as depicted below



We'll repeat the algorithm as described above. First, traverse through the existing tree until we reach a leaf node:



Since this datapoint's `here` percentage is < 0.1, we travel left:



Now we arrive at a leaf node and notice that the label is correct (our model predicts "Spam" as expected by our input). This means we need to make no further changes and can leave our tree as it is!

Repeating this process for all data points in our provided lists will result in a working classification tree trained on existing input data!

ClassificationNode

As part of writing your `ClassificationTree` class, you should also have a **private static inner class** called `ClassificationNode` to represent the nodes of the tree. The contents of this class are up to you, but must meet the following requirements:

- You must have a single `ClassificationNode` class that can represent both splits and labels — you should *not* create separate classes for the different types of nodes.
    - **Don't worry about efficient subclassing/superclassing even though some fields won't be used in all cases. This is entirely ok for this assignment.**
- The fields of the `ClassificationNode` class must be `public`.
- The `ClassificationNode` class must not contain any constructors or methods that are not used by the `ClassificationTree` class.

File Format

The files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent intermediary nodes and a single line to represent leaf nodes in the `ClassificationTree`. The first line in each intermediary node pair will start with "Feature: " followed by the feature and the second line will start with "Threshold: " followed by the threshold. Lines representing the leaf nodes will simply contain the label. The format of the file should be a **pre-order traversal** of the tree.

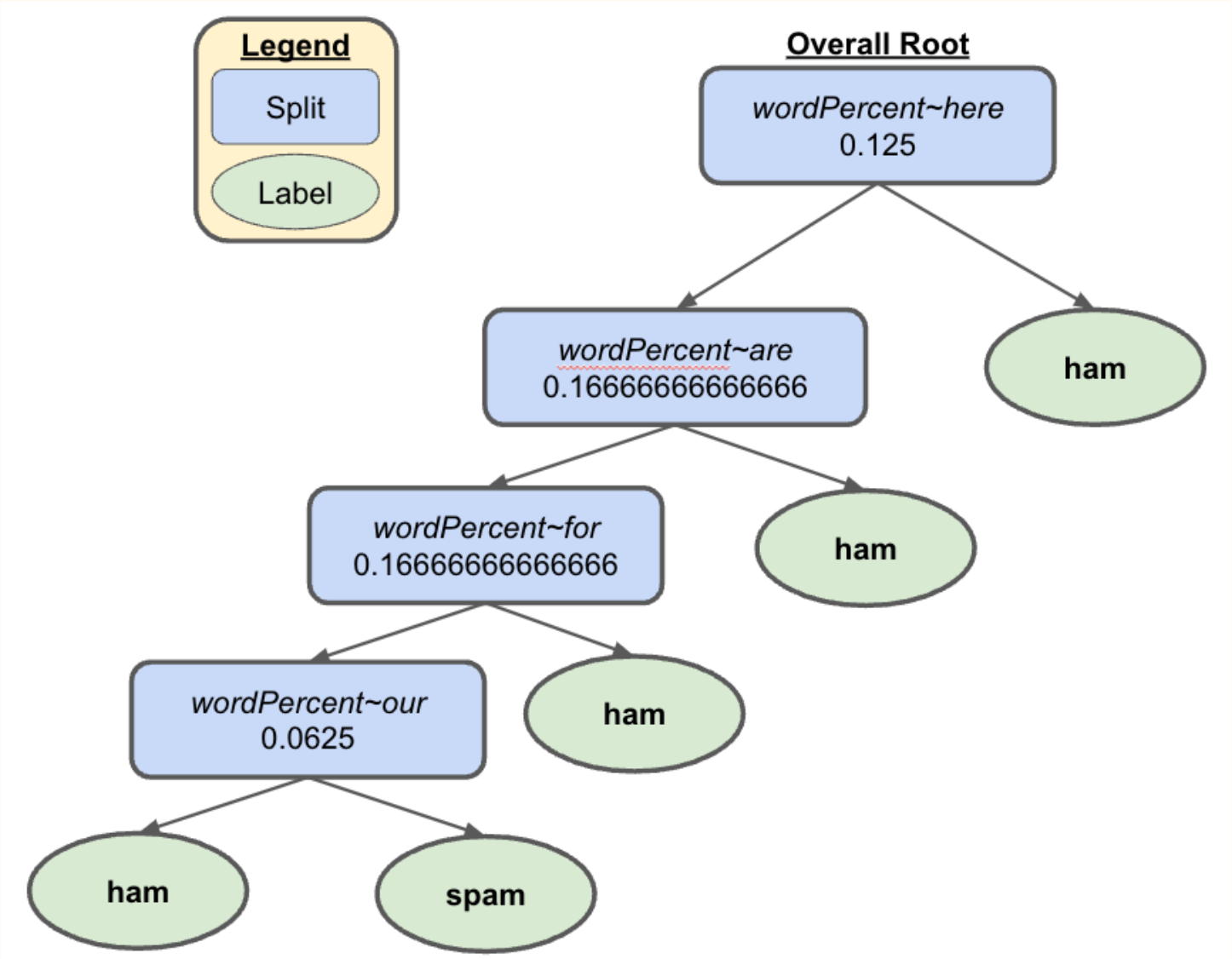
For example, consider the following sample file (`simple.txt`):

▼ Expand

```
Feature: wordPercent~here
Threshold: 0.125
Feature: wordPercent~are
Threshold: 0.16666666666666666
Feature: wordPercent~for
Threshold: 0.16666666666666666
Feature: wordPercent~our
Threshold: 0.0625
ham
spam
ham
ham
```

```
ham
```

Notice that the nodes appear in a ***pre-order traversal*** of the resulting tree:



Try out your Classifier!

Once those methods are implemented, you'll have a working classifier! Try it out using `Client.java` and see how well it does (what is its accuracy on our test data). Also, try saving your tree to a file and see what it looks like. Is it splitting on features you'd expect? Why or why not? (Note that this is a big area of current CS research called "explainable AI" - how can we interpret the results from these massive probability models that are often difficult for humans to understand).

Client Program & Visualization

We have provided you with a `Client` program to help test your implementation of the methods within `ClassificationTree.java`. The client can create binary trees from the provided `.csv` or

`.txt` files and test their accuracy. Note that in order to pass in these files, you should call them by `folderName/fileName`. For example, `trees/simple.txt` .

Click "Expand" below to see sample executions of the client for different situations (user input is **bold and underlined** and additional information is *italicized*).

1. This client visualization uses your Scanner constructor, `calculateAccuracy()`, and `classify()`. The constructor loads a pre-trained model from a given text file. The following inputs allow us to test its accuracy using the pre-set `TEST_FILE` (defined in line 8 of `Client.java`) and use the model to predict labels for data points in a given `test.csv` file.

NOTE: When testing the `Scanner` constructor, the contents of the file will be exactly the same as the input `.txt` file used to initially load the pre-trained model.

▼ Expand

```
Welcome to the CSE 123 Classifier! To begin, enter your desired mode of operation:

1) Train classification model
2) Load model from file
Enter your choice here: 2 (the Scanner constructor)
Please enter the path to the file you'd like to load: trees/simple.txt

What would you like to do with your model?

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 2 (calculateAccuracy())
Overall: 0.8637632607481853
ham: 0.9961365099806826

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 1 (classify() called on every data point)
Please enter the file you'd like to test: weather/test.csv
Results: [ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, h
ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, h
ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, h
...
ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, ham, h

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 4
```

2. This client visualization uses the `ClassificationTree` constructor that takes in data and their corresponding labels, `calculateAccuracy()` (implemented for you), and `save()`. You can follow the pattern of inputs below to train the classification model using some `train.csv` file (this calls the constructor), retrieve testing accuracy (similar to above) and save the trained model to a file so that it is in `.txt` format (like the sample input files in the trees/ folder).

▼ Expand

```
Welcome to the CSE 123 Classifier! To begin, enter your desired mode of operation:

1) Train classification model
2) Load model from file
Enter your choice here: 1 (data/results constructor)

What would you like to do with your model?

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 2 (calculateAccuracy())
Overall: 0.5713753954959985
ham: 0.5556986477784932
spam: 0.6736694677871149

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 3 (save())
Please enter the file name you'd like to save to: destinationFile.txt

1) Test with an input file
2) Get testing accuracy
3) Save to a file
4) Quit
Enter your choice here: 4
```

NOTE: After quitting, the saved file should be available for viewing in the console.

Testing

There are no formal testing requirements for this assignment. However, we'd encourage you to get your hands dirty and see how well your model performs on the provided dataset / investigate the output files to see if you can make sense of what the inner structure is!

Implementation Guidelines

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements:

- **Constructors in inner class:**
  - Any constructors created should be used.
  - When applicable, reduce redundancy by using the `this()` keyword to call another constructor in the same class.
  - Clients of the class should never have to set fields of an object unconditionally after construction — there should be a constructor included for this situation.
- **Methods:**
  - All methods present in `ClassificationTree` that are not listed in the specification must be `private`.
  - Make sure that all parameters within a method are used and necessary.
  - Avoid unnecessary returns.
- **`x = change(x)`:**
  - Similar to with linked lists, do not "morph" a node by directly modifying fields (especially when replacing an intermediary node with a leaf node or vice versa). Existing nodes can be rearranged in the tree, but adding a new value should always be done by creating and inserting a new node, not by modifying an existing one.
  - An important concept introduced in lecture was called `x = change(x)`. This idea is related to the proper design of recursive methods that manipulate the structure of a binary tree. **You should follow this pattern where necessary when modifying your trees**.
- **Avoid redundancy:**
  - If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here.
  - Look out for including additional base or recursive cases when writing recursive code. While multiple calls may be necessary, you should avoid having more cases than you need. Try to see if there are any redundant checks that can be combined!
- **Data Fields:**
  - Properly encapsulate your objects by making data fields in your `ClassificationTree` class private. (Fields in your `ClassificationNode` class should be public following the pattern from class.)
  - Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place.
  - Fields should always be initialized inside a constructor or method, never at declaration.
- **Commenting**
  - Each method should have a comment including all necessary information as described in

the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec).

- Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
- Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.