Programming Assignment 3: Cornbear's Classifier

Background and Structure

Seemingly, everyone is talking about Machine Learning, Artificial Intelligence and Cornbear these days. Artificial Intelligence (AI), a subfield of Computer Science, is concerned with enabling computers to perform tasks that require rational decision-making. As one of the oldest areas of research in the discipline, AI has played a significant role in driving technological advancements since the 1950s. On the other hand, machine Learning (ML) is a subfield of AI that uses trends from previous examples to make predictions 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 plain 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.

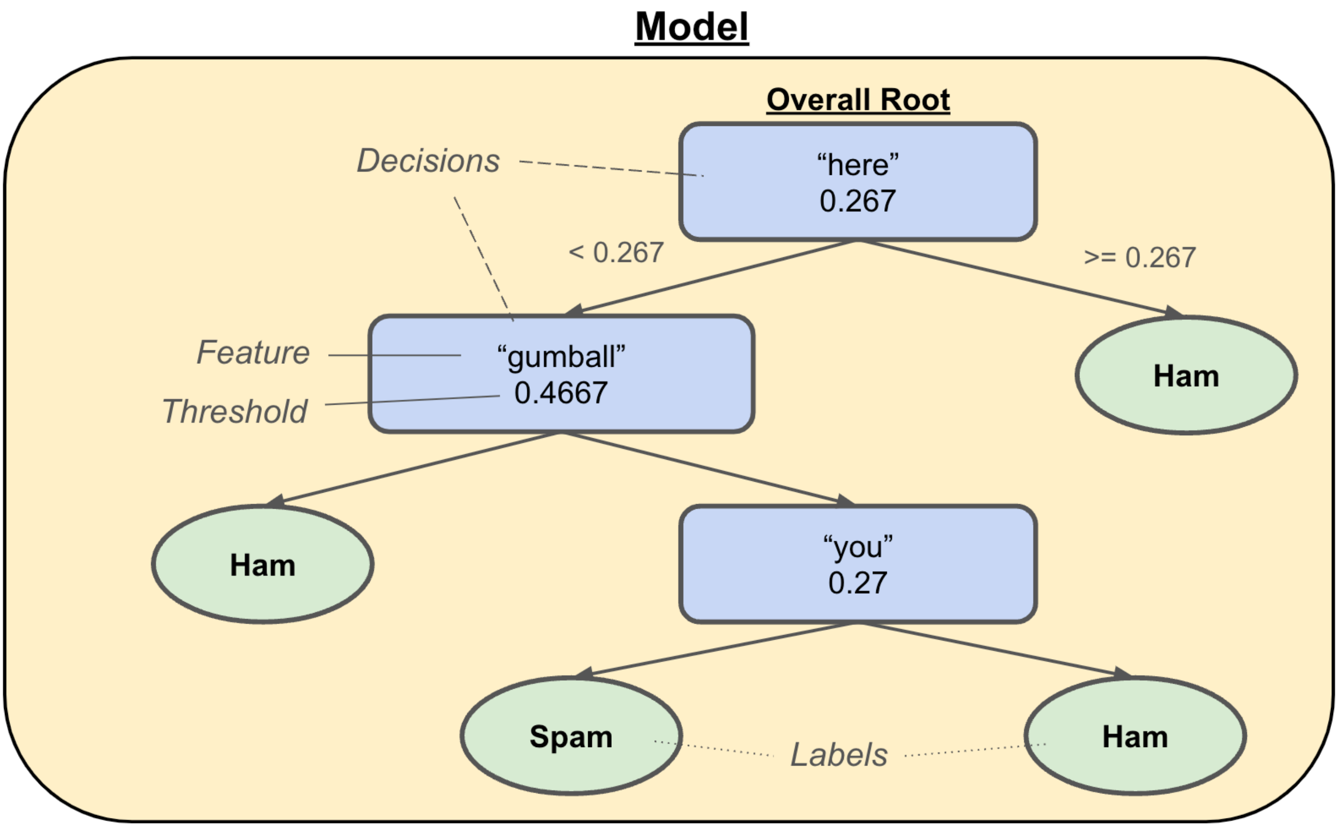
- **Model:** The actual program that makes probabilistic classifications on provided inputs.
- **Training:** Models are "trained" on previously gathered datasets to make future predictions.
- **Label**: How data is classified after being run through the model. In our tree, leaf nodes will house classification labels.
- **Feature**: Features are important and measurable properties of an item in our data set that help classify our data to make quality predictions. In our tree, the features will be represented as the probabilities of each word from our sentences.

- **Threshold**: The numeric value we compare a feature against at any branch node 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.

Structure

Your goal for this assignment is to implement machine-learning model classifier for everyone's favorite mascot, Cornbear!!! Specifically, you will be implementing a text-based classification tree, a (relatively) simplistic machine-learning model that predicts a label when given some text-based data. In this section, we'll familiarize you with the classifier's visual structure. Additionally, this assignment involves a lot of Machine Learning (ML) terminology. For clarity, these terms are underlined within this specification

Below is a visual example of what a classification tree might look like for classifying spam emails:



Expand to see an alternate text representation of the above classification tree:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.267
  - "gumball" (left) (level 1) decision node, threshold = 0.4667
    - Ham (left) (level 2) label node
    - "you" (right) (level 2) decision node, threshold = 0.27
      - Spam (left) (level 3) label node

        - Ham (right) (level 3) label node
    - Ham (right) (level 1) label node

In our classification tree, the **leaf nodes represent our predictive <u>labels</u>** ("Spam" or "Ham" – a funny way of writing not spam) while the **branch nodes represent decision nodes** that contains some <u>feature</u> of our data and a <u>threshold</u> to determine what decision to make. For this assignment, the <u>feature</u> will be the word probability of a certain word.

As mentioned earlier, you will be given text-based data to classify. This may include, but is not limited to, emails, academic papers, or even movie reviews! Throughout this assignment, each piece of text will be called **text blocks**, and we'll represent them with the `TextBlock` class. (more on that in the Implementation Requirements slide).

To classify a given text block, you start at the root of the tree and determine whether the corresponding <u>feature</u> found in the input text block falls to the left or right of the current node's <u>threshold</u> (determined by `<` or `>=`). Then, you travel in the corresponding direction. Repeating this process will eventually lead you to a classification for your input.

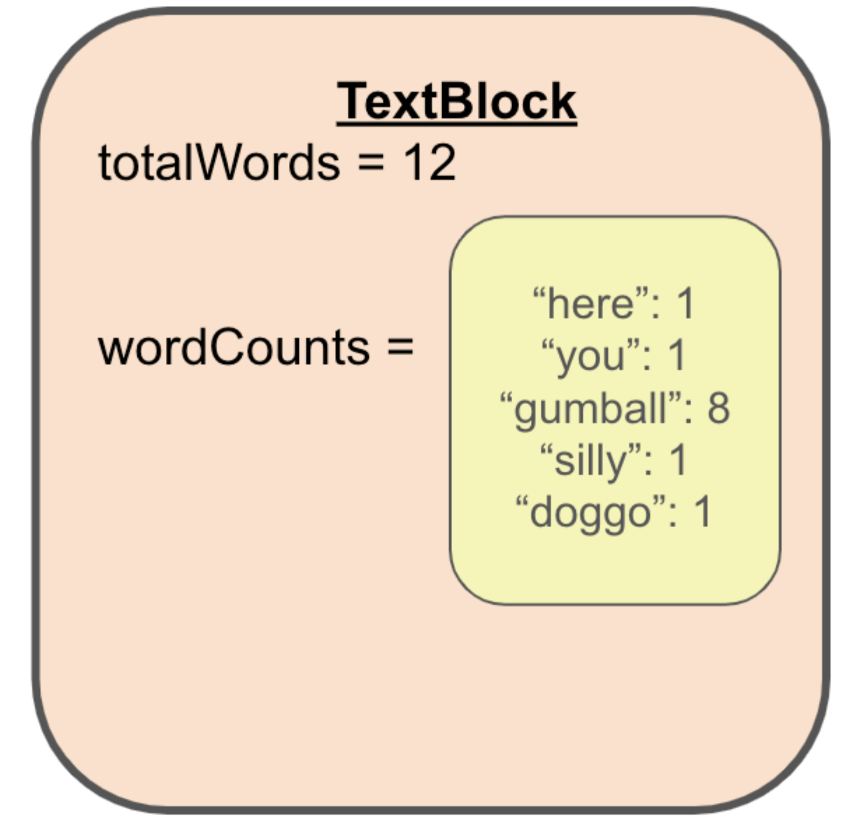
Below, we'll trace through the classification of a sample input with our example model shown above.

▼ Expand

We'll begin at the root node with a `TextBlock` object called (**"Input"**) created on the following text:

```
here gumball gumball gumball gumball you silly gumball gumball gumball gumball doggo
```

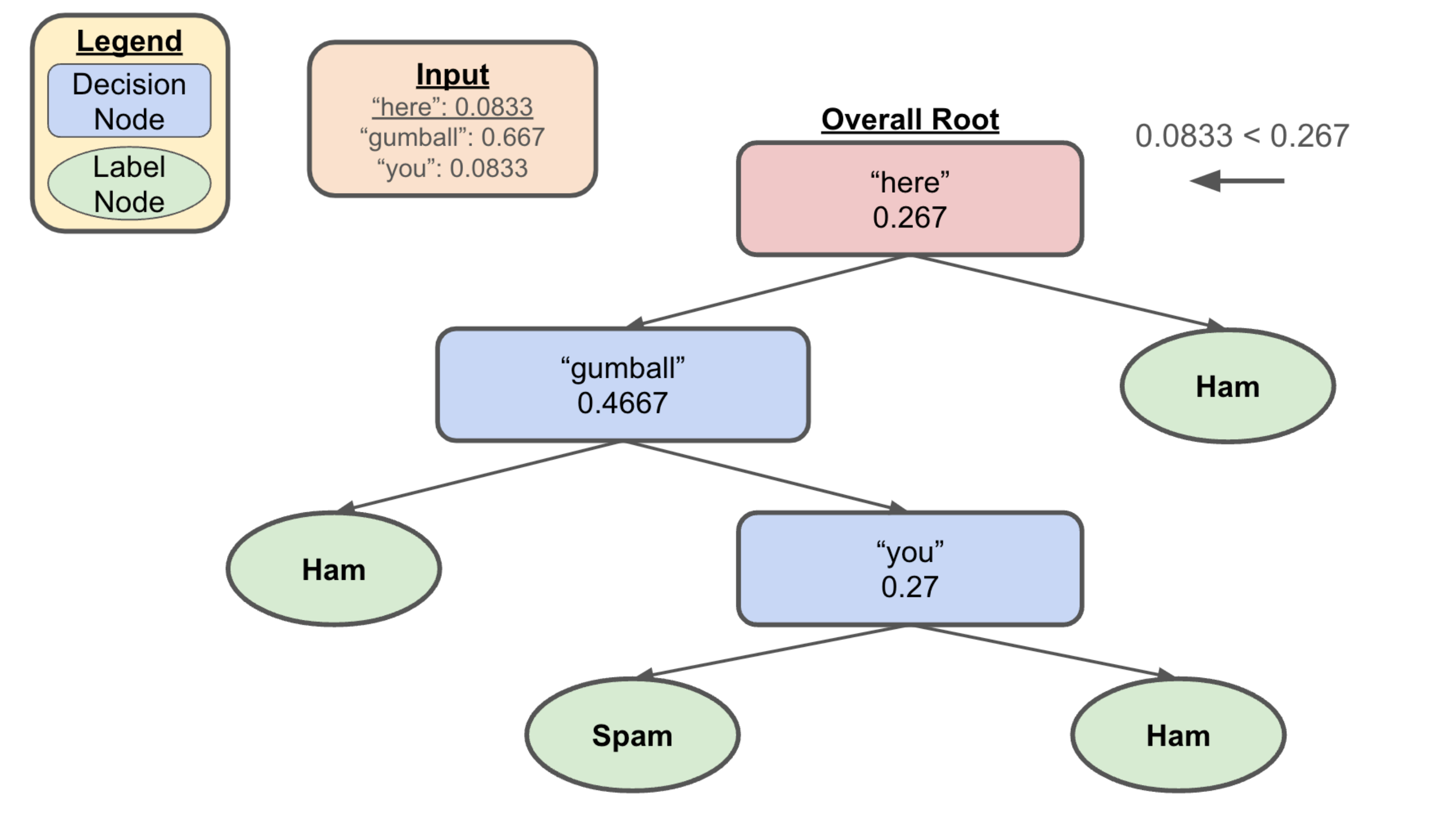
producing the following `TextBlock` object:



In our `TextBlock` object, it has `totalWords=12`, as well as a mapping `wordCounts` of a word to its count. In this example, `"here"`, `"you"`, `"silly"`, and `"doggo"` occurs once in the content, hence each of these words are mapped to `1`. However "gumball" appears eight times in the provided content, thus `"gumball"` is mapped to `8`.

Note that our classifier uses word probability rather than word frequency, so the appropriate conversions would be 0.667 (`8/12`) for `"gumball"` and 0.0833 (`1/12)` for `"here"`, `"you"`, `"silly"`, and `"doggo"`.

1. Since the word probability of `"here"` in our `TextBlock` (0.0833) is less than the threshold (0.267), we travel left of the `"here"` node to the `"gumball"` node.



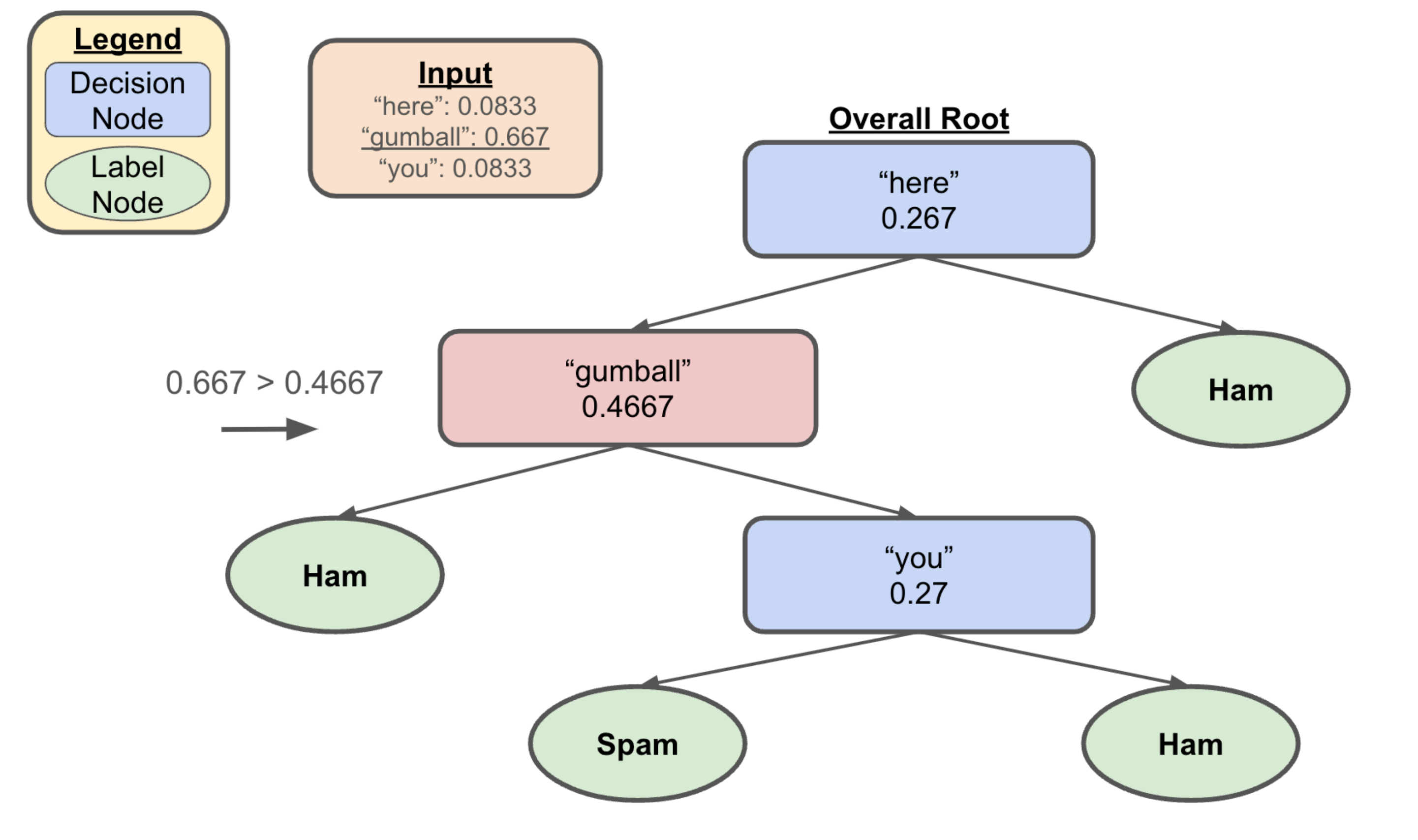
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.267
  - "gumball" (left) (level 1) decision node, threshold = 0.4667
    - Ham (left) (level 2) label node
    - "you" (right) (level 2) decision node, threshold = 0.27
      - Spam (left) (level 3) label node
      - Ham (right) (level 3) label node
  - Ham (right) (level 1) label node

2. Since the value of the `"gumball"` feature in our `TextBlock` (0.667) is greater than or equal to the threshold (0.4667), we'll travel right of the `"gumball"` node to the `"you"` node



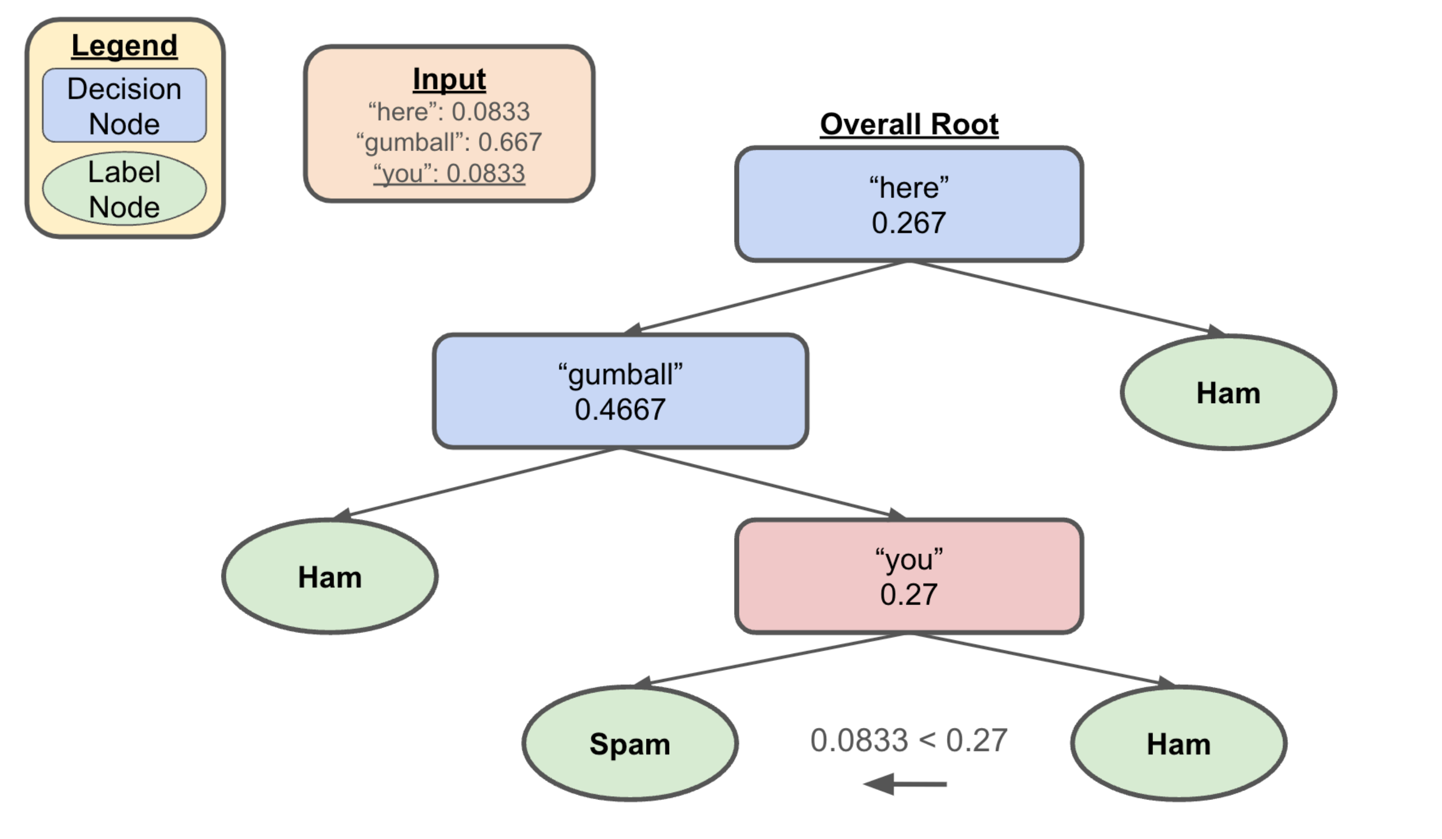
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.267
  - **current node:** "gumball" (left) (level 1) decision node, threshold = 0.4667
    - Ham (left) (level 2) label node
    - "you" (right) (level 2) decision node, threshold = 0.27
      - Spam (left) (level 3) label node
      - Ham (right) (level 3) label node
  - Ham (right) (level 1) label node

3. Since the value of the `"you"` feature (0.0833) is less than the threshold (0.27) we'll travel left of the `"you"` node to the `Spam` node



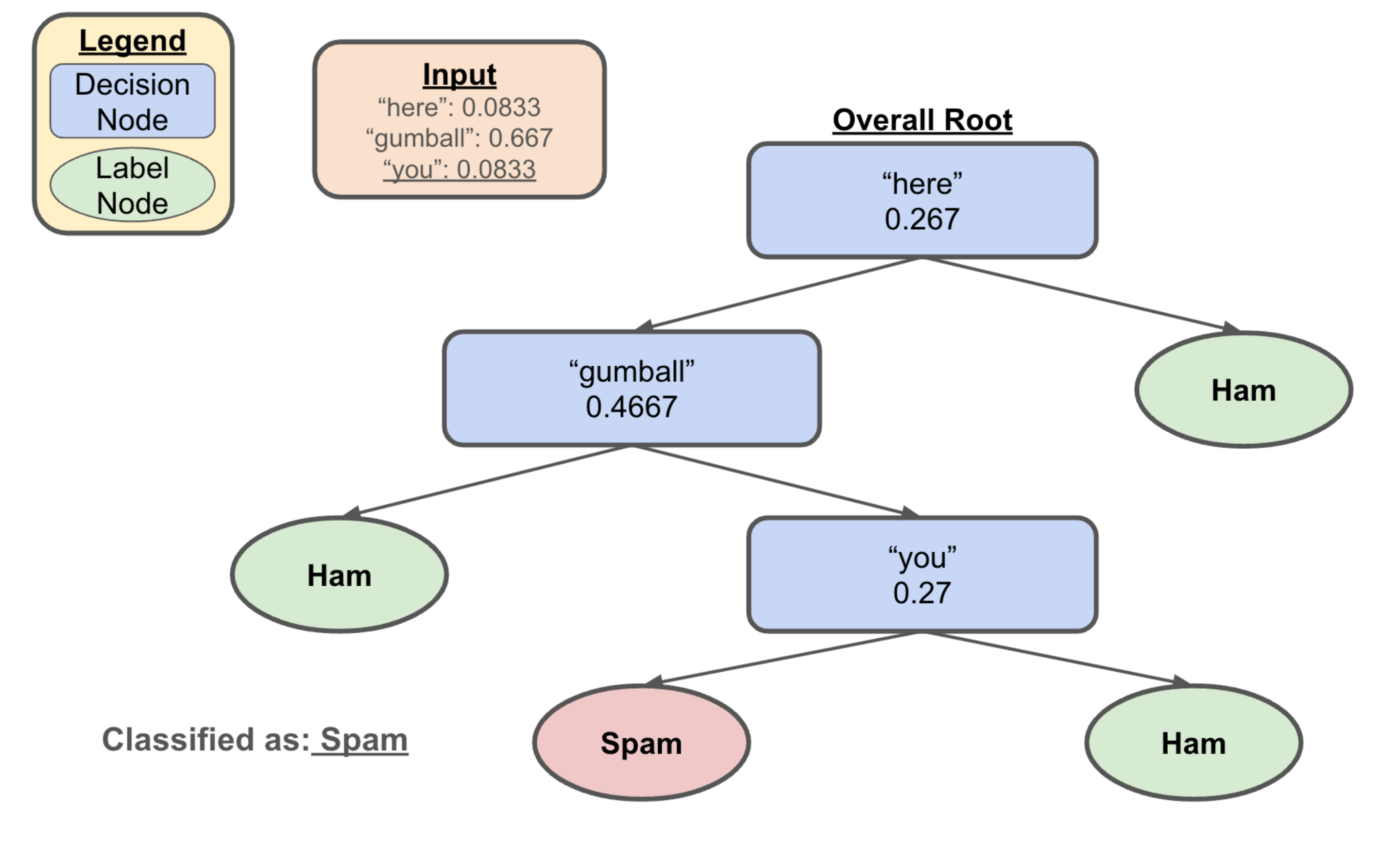
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.267
  - "gumball" (left) (level 1) decision node, threshold = 0.4667
    - Ham (left) (level 2) label node
    - **current node** "you" (right) (level 2) decision node, threshold = 0.27
      - Spam (left) (level 3) label node
      - Ham (right) (level 3) label node
  - Ham (right) (level 1) label node

4. We have reached a leaf node and therefore can predict that our input corresponds to "Spam" — a spam email (the resulting label)



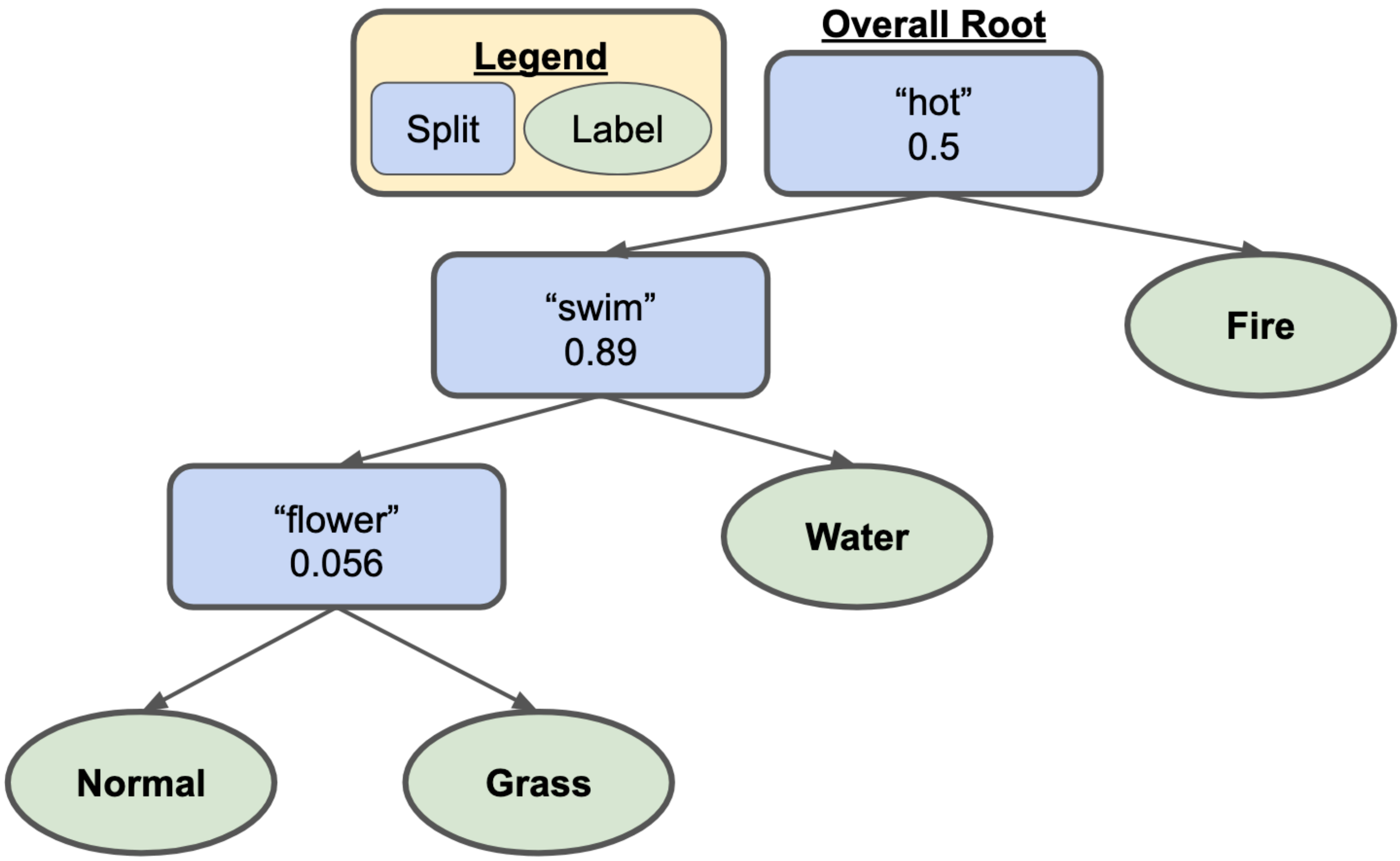
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.267
  - "gumball" (left) (level 1) decision node, threshold = 0.4667
    - Ham (left) (level 2) label node
    - "you" (right) (level 2) decision node, threshold = 0.27
      - **current node** Spam (left) (level 3) label node
      - hHm (right) (level 3) label node
  - Ham (right) (level 1) label node

These classification trees may not always be the same, and may not always operate on identifying `"spam"` or `"ham"`. Below is an alternative example of what a potential classification tree could look like for Pokémon types based on text from Pokedex entries.



Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

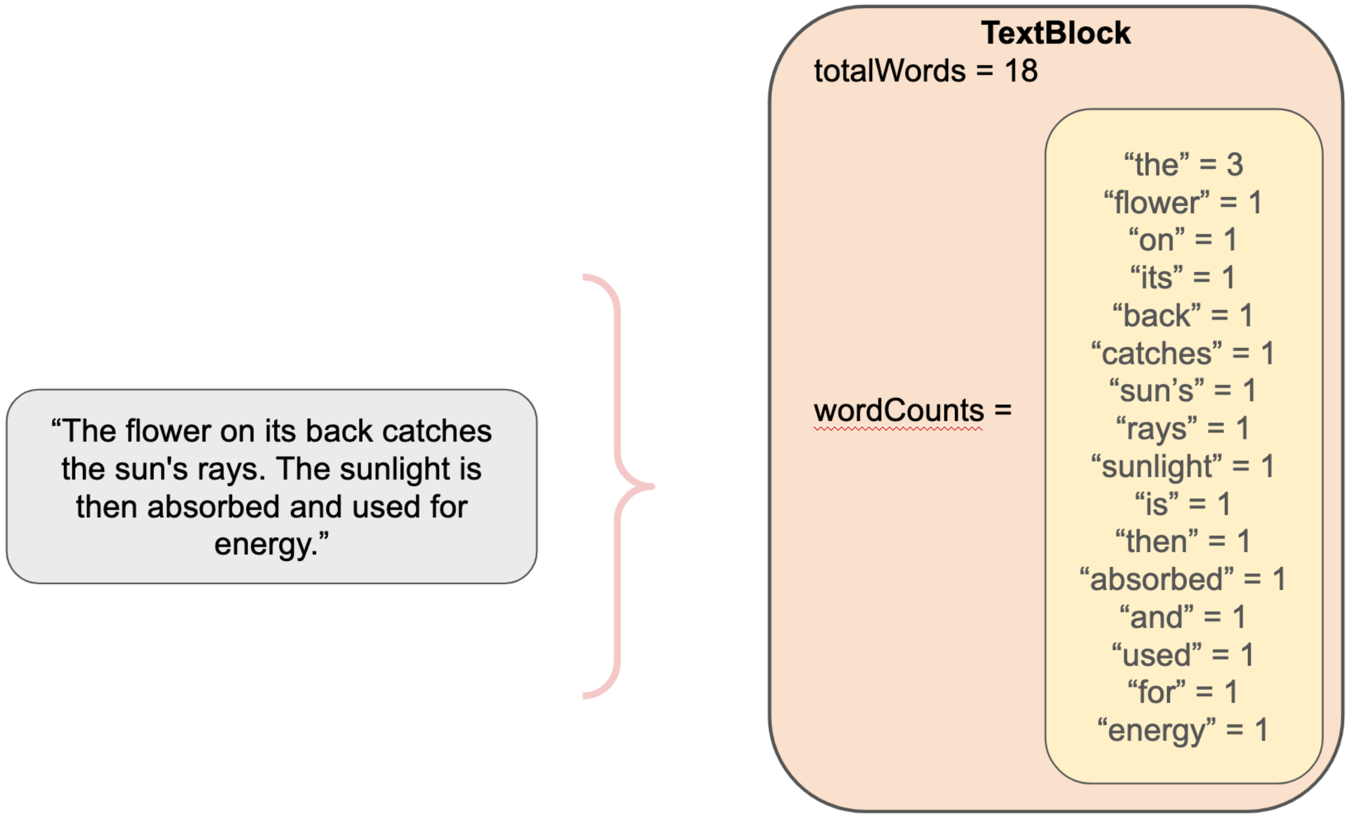
Classification Tree:

- "hot" (root) (level 0) decision node, threshold = 0.5
    - "swim" (left) (level 1) decision node, threshold = 0.89
        - "flower" (left) (level 2) decision node, threshold = 0.056
            - Normal (left) (level 3) label node
            - Grass (right) (level 3) label node
        - Water (right) (level 2) label node
    - Fire (right) (level 1) label node

To solidify the different tree behaviors, we'll trace through an input much like the example above.

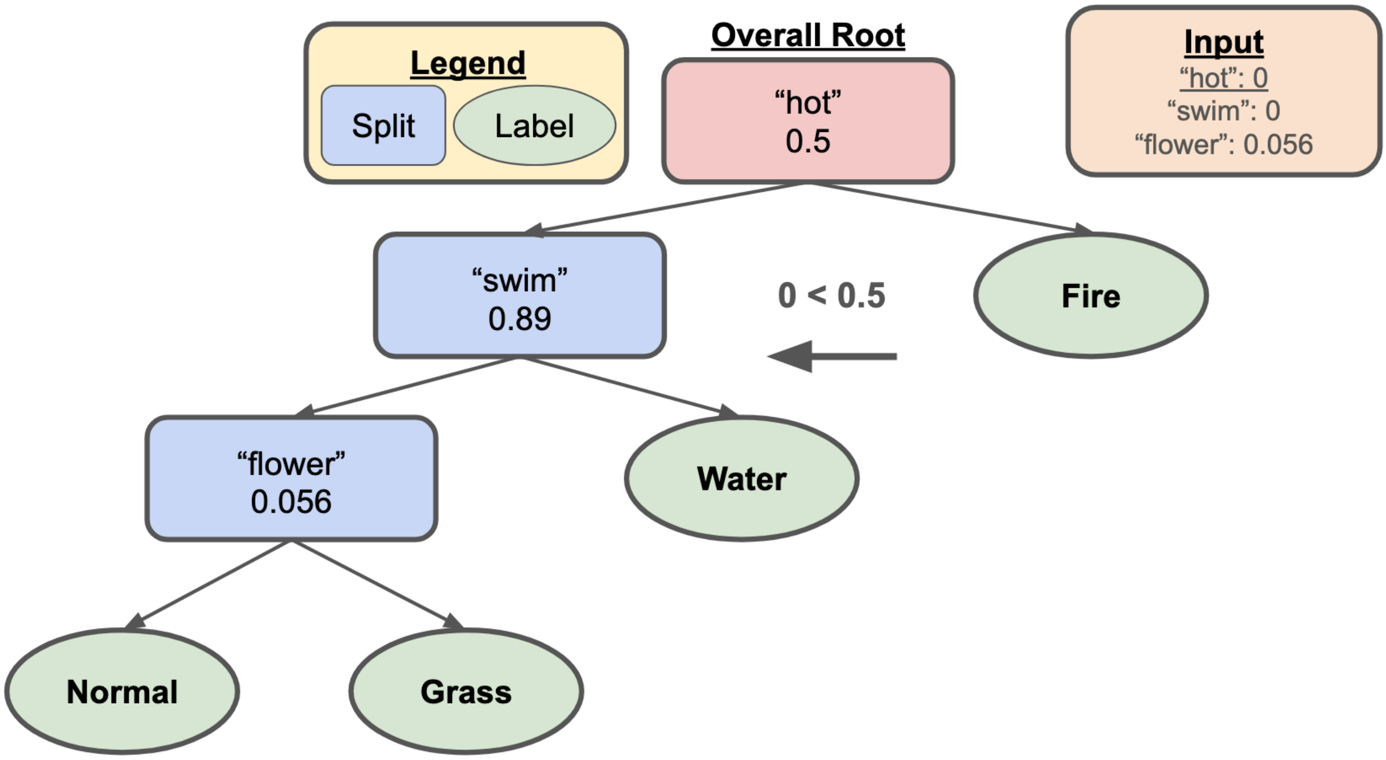
▼ Expand

We'll begin at the root node with the following input from Venusaur's Pokedex entry, which is "The flower on its back catches the sun's rays. The sunlight is then absorbed and used for energy":



In this example, the total number of words is 18, with most words having a `wordCount` of 1, except `"the"`, which has a `wordCount` of 3. As a **reminder**, words not present in the input implicitly have a `wordCount` of 0.

1. Since the word `"hot"` is not present in the entry, it has a word probability of 0, which is less than the threshold (0.5), we'll travel left of the `"hot"` node to the `"swim"` node.



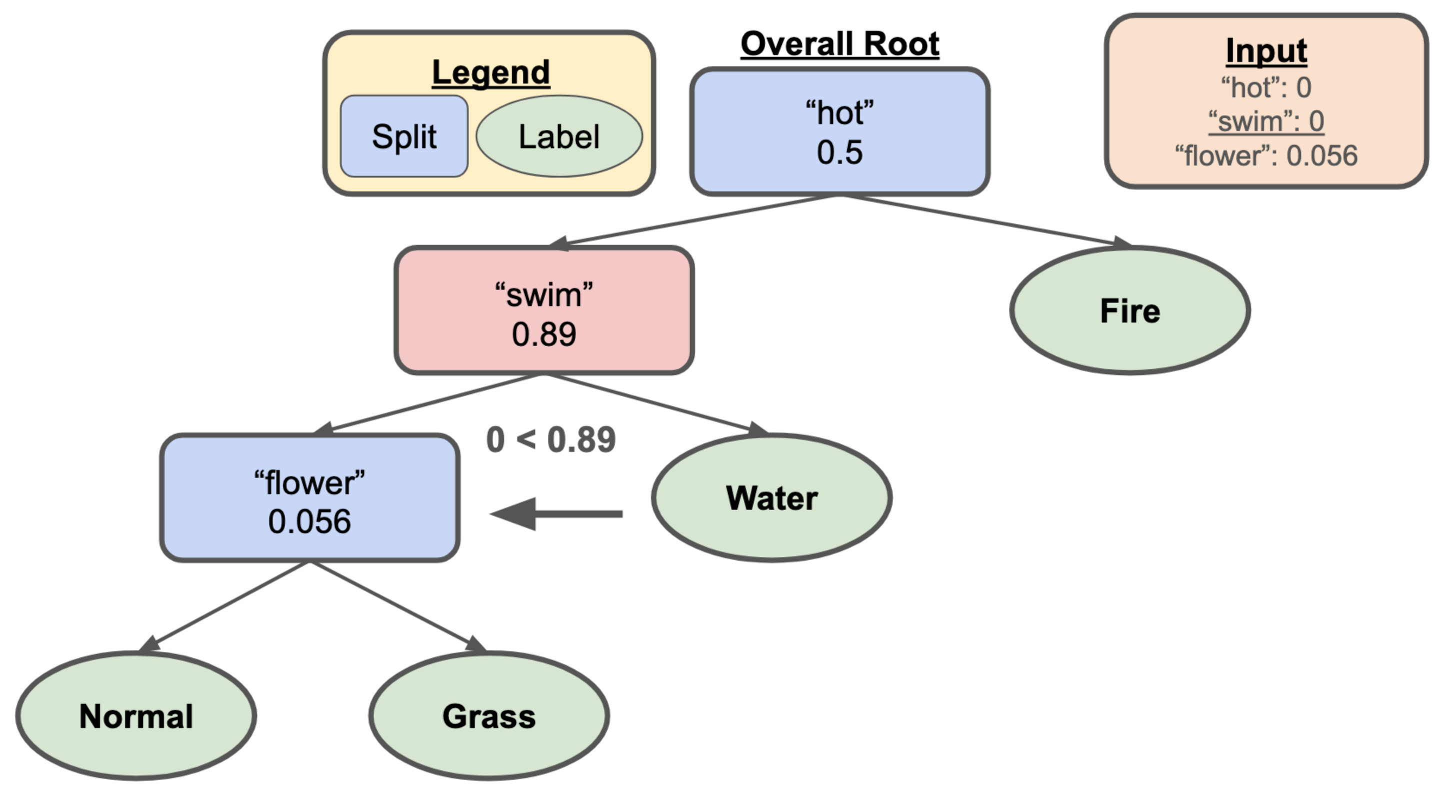
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- **current node:** "hot" (root) (level 0) decision node, threshold = 0.5
  - "swim" (left) (level 1) decision node, threshold = 0.89
    - "flower" (left) (level 2) decision node, threshold = 0.056
      - Normal (left) (level 3) label node
      - Grass (right) (level 3) label node
    - Water (right) (level 2) label node
  - Fire (right) (level 1) label node

2. Similarly, since the word `"swim"` is not present, it has a word probability of 0, which is less than the threshold (0.89), we'll travel left of the `"swim"` node to the `"flower"` node.



Expand to see an alternate equivalent representation of the above classification tree:

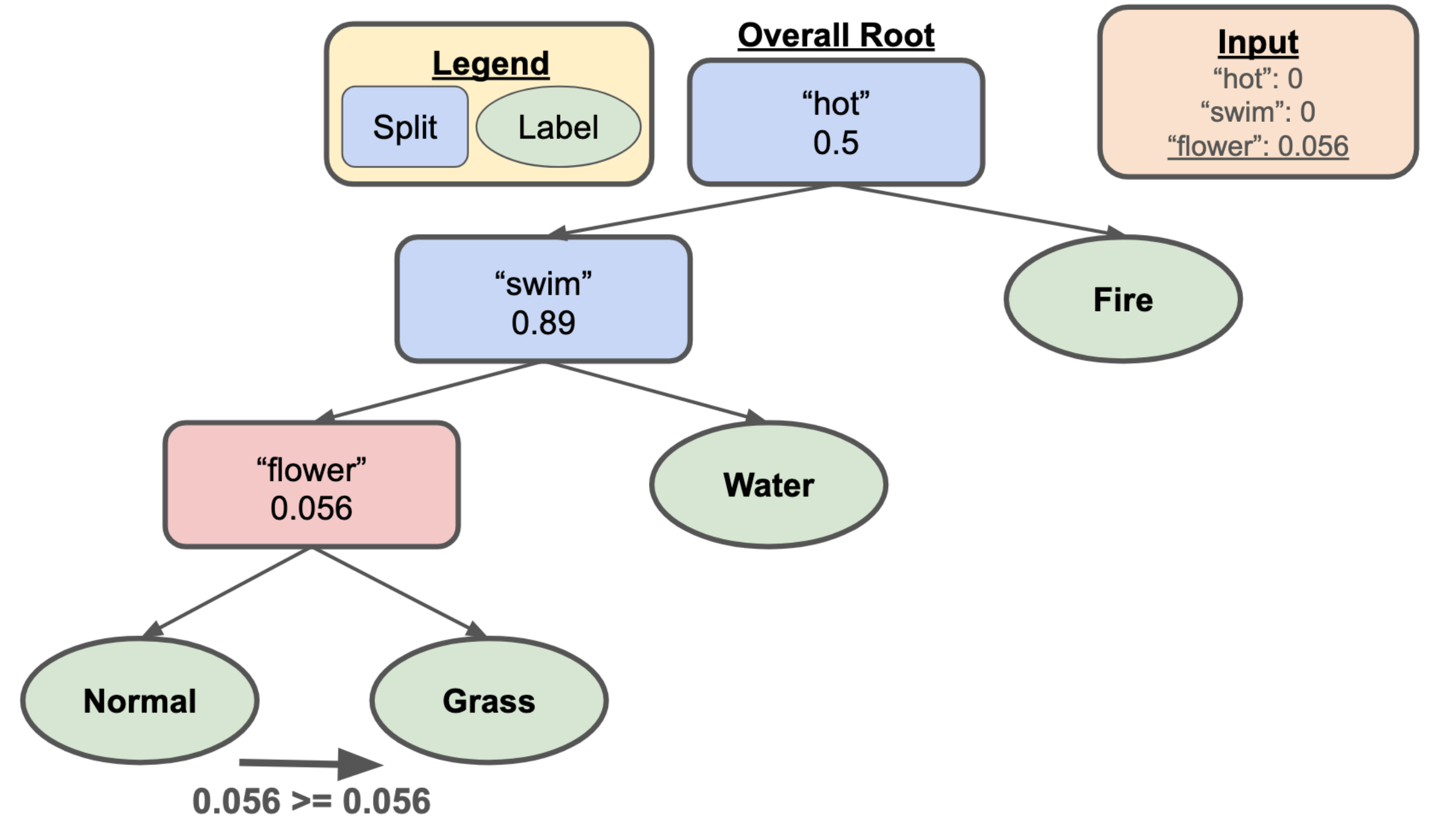
▼ Expand

Classification Tree:

- "hot" (root) (level 0) decision node, threshold = 0.5
  - **current node** "swim" (left) (level 1) decision node, threshold = 0.89
    - "flower" (left) (level 2) decision node, threshold = 0.056
      - Normal (left) (level 3) label node

        - Grass (right) (level 3) label node
      - Water (right) (level 2) label node
    - Fire (right) (level 1) label node

3. Since the word probability of `"flower"` is 0.056, which is greater than or equal to the threshold (0.056), we'll travel right to the `Grass` node.



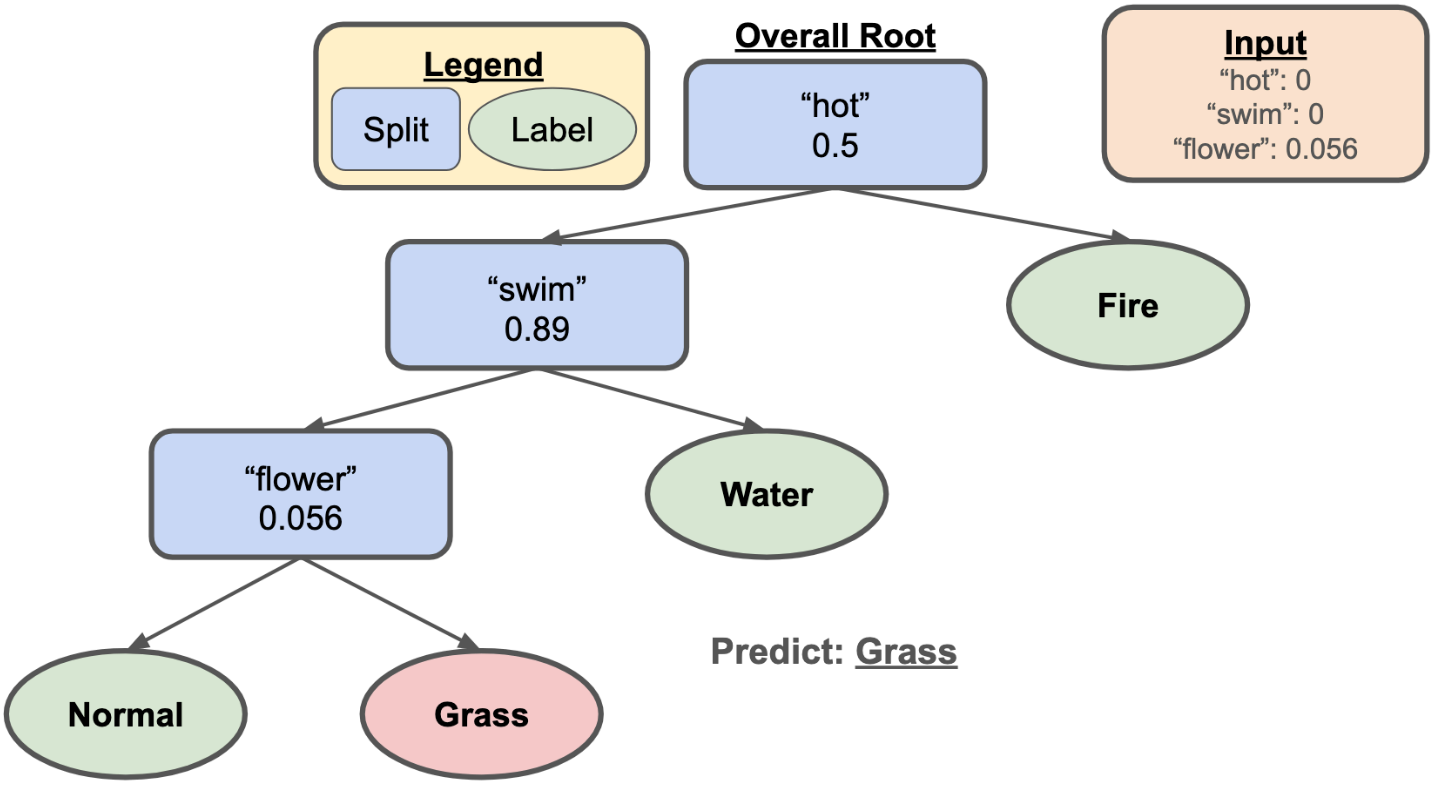
Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- "hot" (root) (level 0) decision node, threshold = 0.5
    - "swim" (left) (level 1) decision node, threshold = 0.89
      - **current node:** "flower" (left) (level 2) decision node, threshold = 0.056
        - Normal (left) (level 3) label node
        - Grass (right) (level 3) label node
      - Water (right) (level 2) label node
    - Fire (right) (level 1) label node

4. We have reached a leaf node and therefore can predict that our input corresponds to a Grass-type Pokémon (the resulting label).



Expand to see an alternate equivalent representation of the above classification tree:

▼ Expand

Classification Tree:

- "hot" (root) (level 0) decision node, threshold = 0.5
  - "swim" (left) (level 1) decision node, threshold = 0.89
    - "flower" (left) (level 2) decision node, threshold = 0.056
      - Normal (left) (level 3) label node
      - **current node:** Grass (right) (level 3) label node
    - Water (right) (level 2) label node
  - Fire (right) (level 1) label node

This is what you'll be implementing in this assignment! Specifically, you'll be creating a classification tree that's able to predict a label given some text. This could range from predicting "Spam" or "Ham" given the contents of an email (as shown above) to predicting the author of a given Federalist Paper!

Training a Classification Tree

One of our goals is to be able to "train" our model from previously gathered data in order to make future predictions.

In the previous slide, we magically arrived at a constructed classification tree. In this section, we'll explain the algorithm to train a new model. Throughout this section, we'll be using the following file (which can be found in `spec_example.csv`):

```
Category,Message
Ham,here here here four five six seven eight nine ten eleven tweleve thirteen office you
Spam,one two three four five six seven eight nine ten eleven tweleve thirteen office you
Spam,one two three four five six seven eight nine ten eleven tweleve thirteen office you
Spam,here here here office office office office office office office office office office of
```

Note the structure of this `.csv` file: the second column contains the data, and the first column contains the expected label for that piece of data. For ease of implementation, this file will automatically be parsed for you (using the provided `DataLoader` and `CsvReader` classes) and passed into your constructor as two lists.

```
public Classifier(List<TextBlock> data, List<String> labels)
```

Step 1: Initialize our Model

Since the classification tree is empty at the beginning, we need to add an initial data point so it can start making classifications. This algorithm processes training examples in order, starting from index `0`. So we begin by inserting the first data-label pair (at index 0) into the tree.

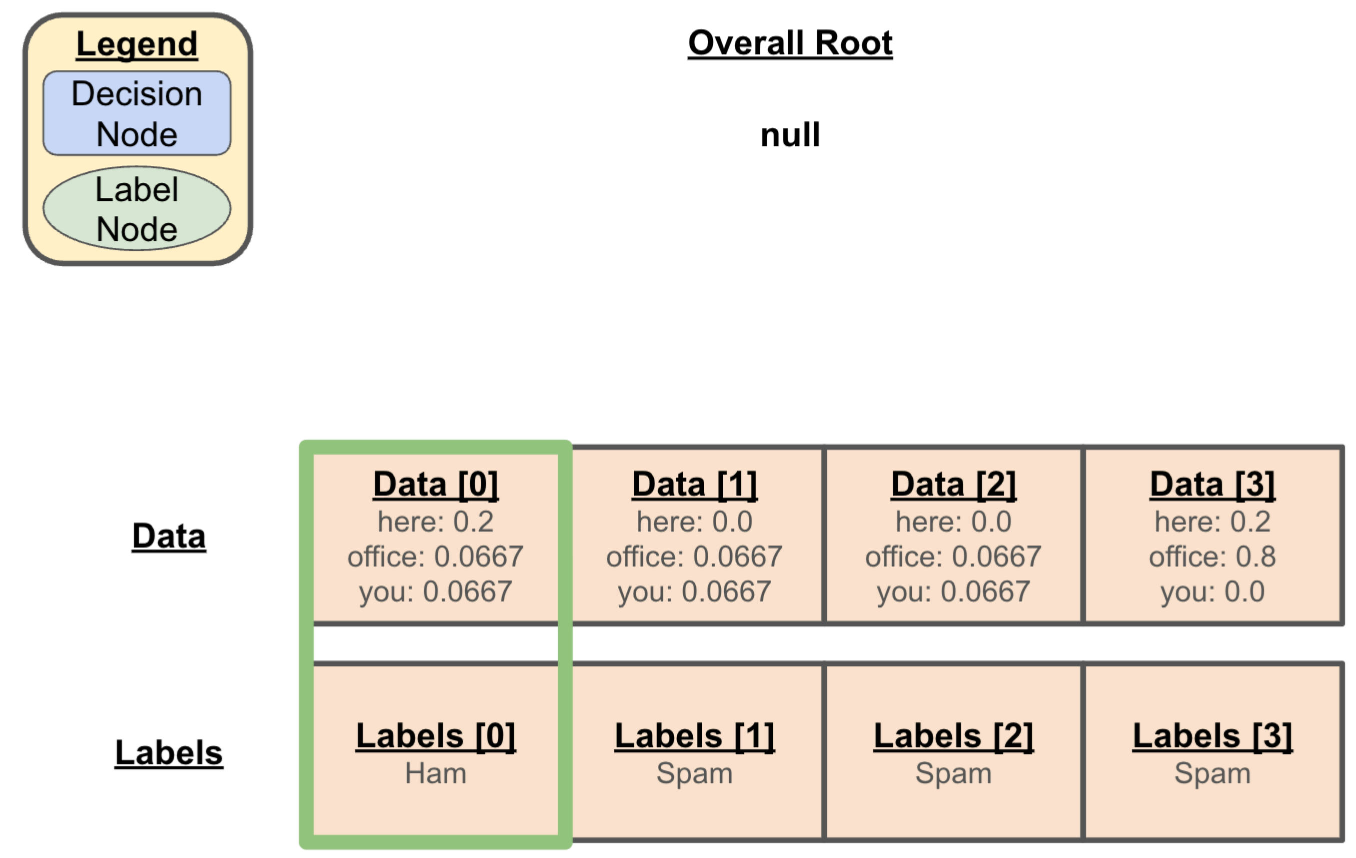
We also want to store the `TextBlock` data along with its label in the tree's leaves. This way, the classification tree can use previous examples to help make decisions when creating new nodes. If this part isn't entirely clear yet, that's okay. We will see this action later in the algorithm explanation.

Expand to see the visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We process index `0`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- null (root)

Data List:

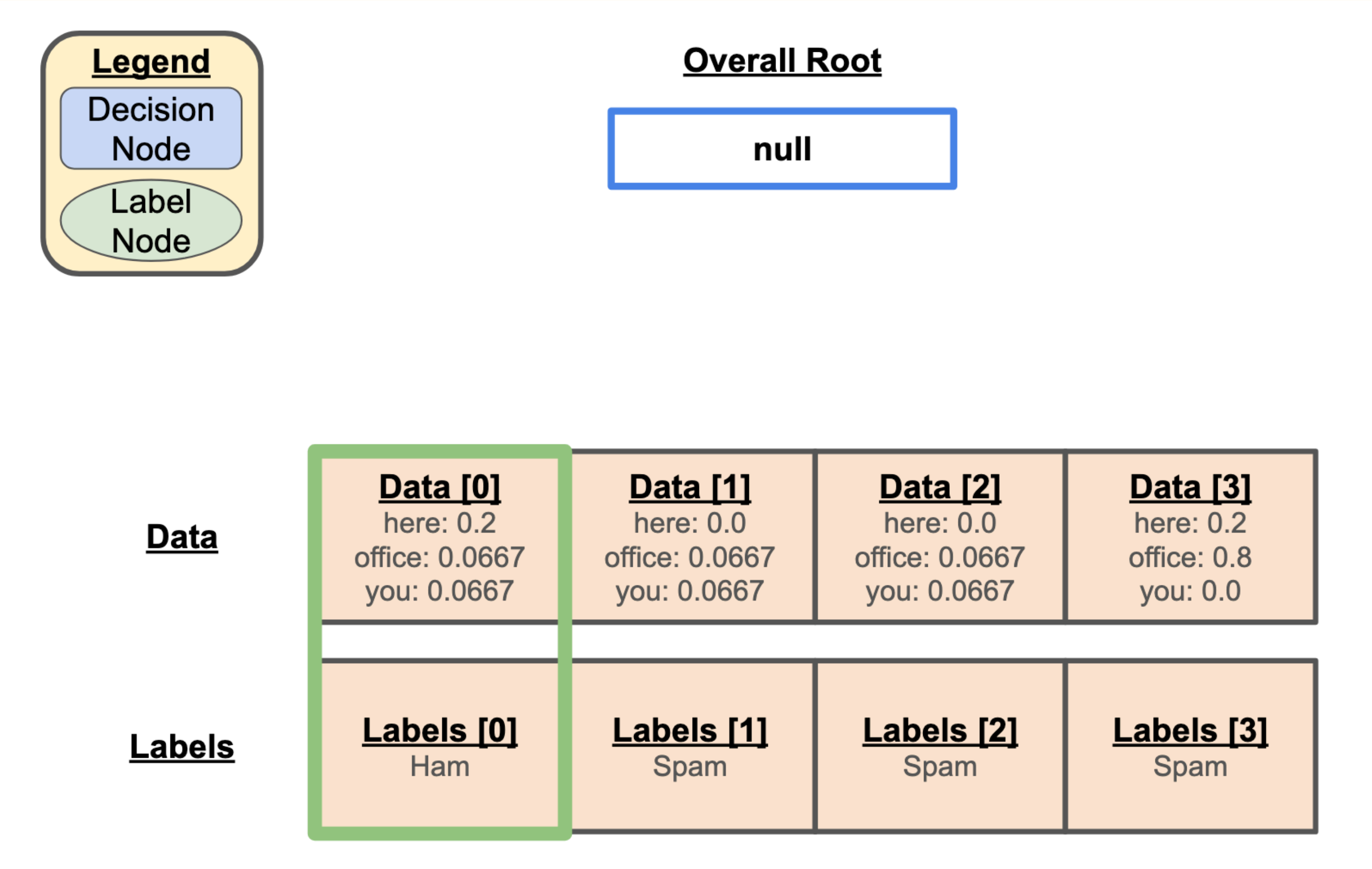
- **Current item:** Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8

- word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

The current tree is empty.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** null (root)

Data List:

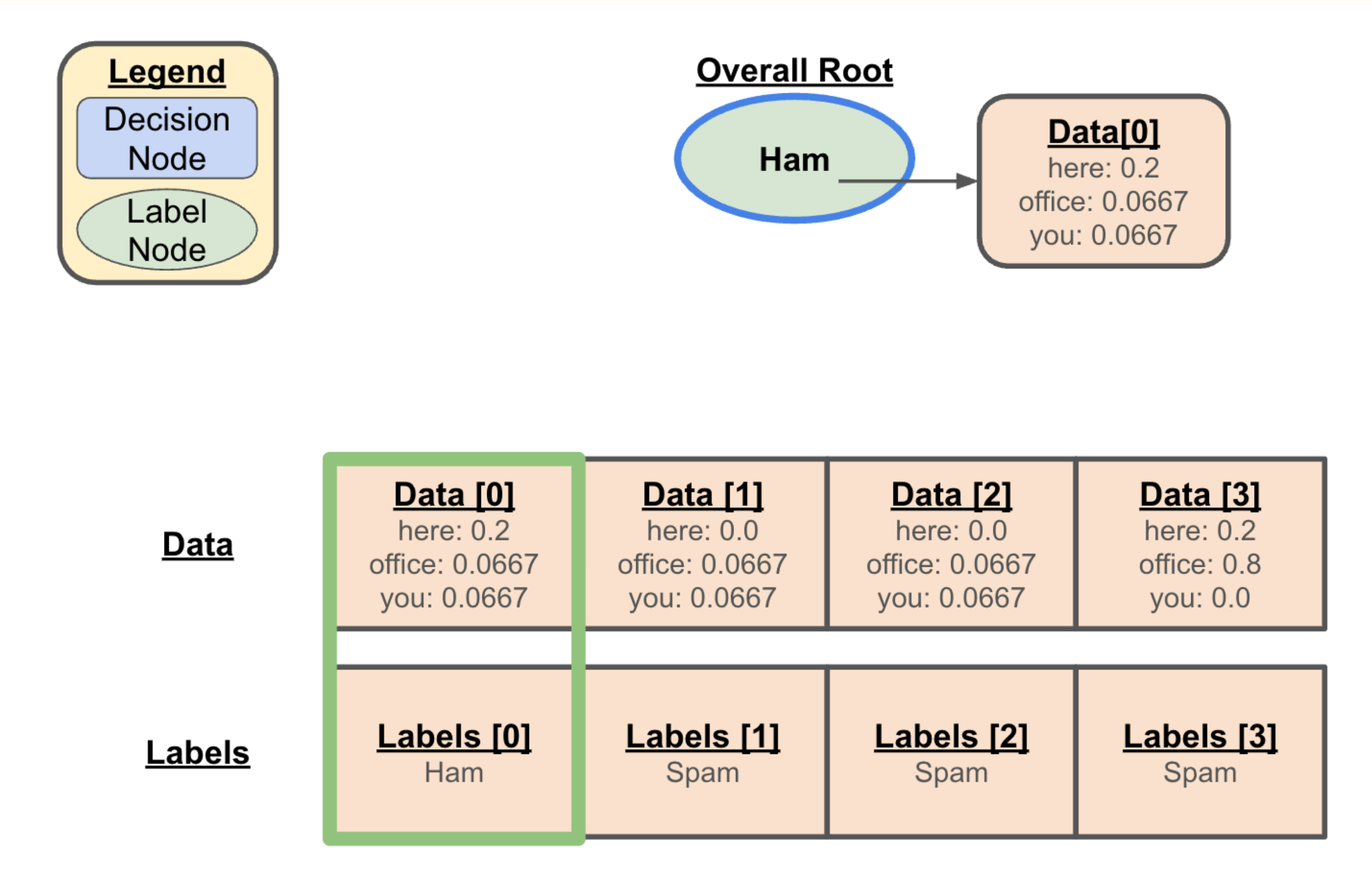
- **Current item:** Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:

- word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

We create a new label node to store the information at index `0` since the tree is currently empty.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0])

Data List:

- **Current item:** Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Step 2: Classify Data

Now that we have a classification tree, we can start to classify inputs! Unfortunately, with only one point of data, our model doesn't seem very useful — currently, it classifies every input as `Ham`. What if we try to classify a piece of data that has an expected label of `Spam`?

To handle this, we proceed to the next step of the algorithm: we process the next index. We'll **start at the top of the tree** and traverse down to find the label our model will predict for `data.get(index)`. Now, we check whether our model's prediction matches the expected label.

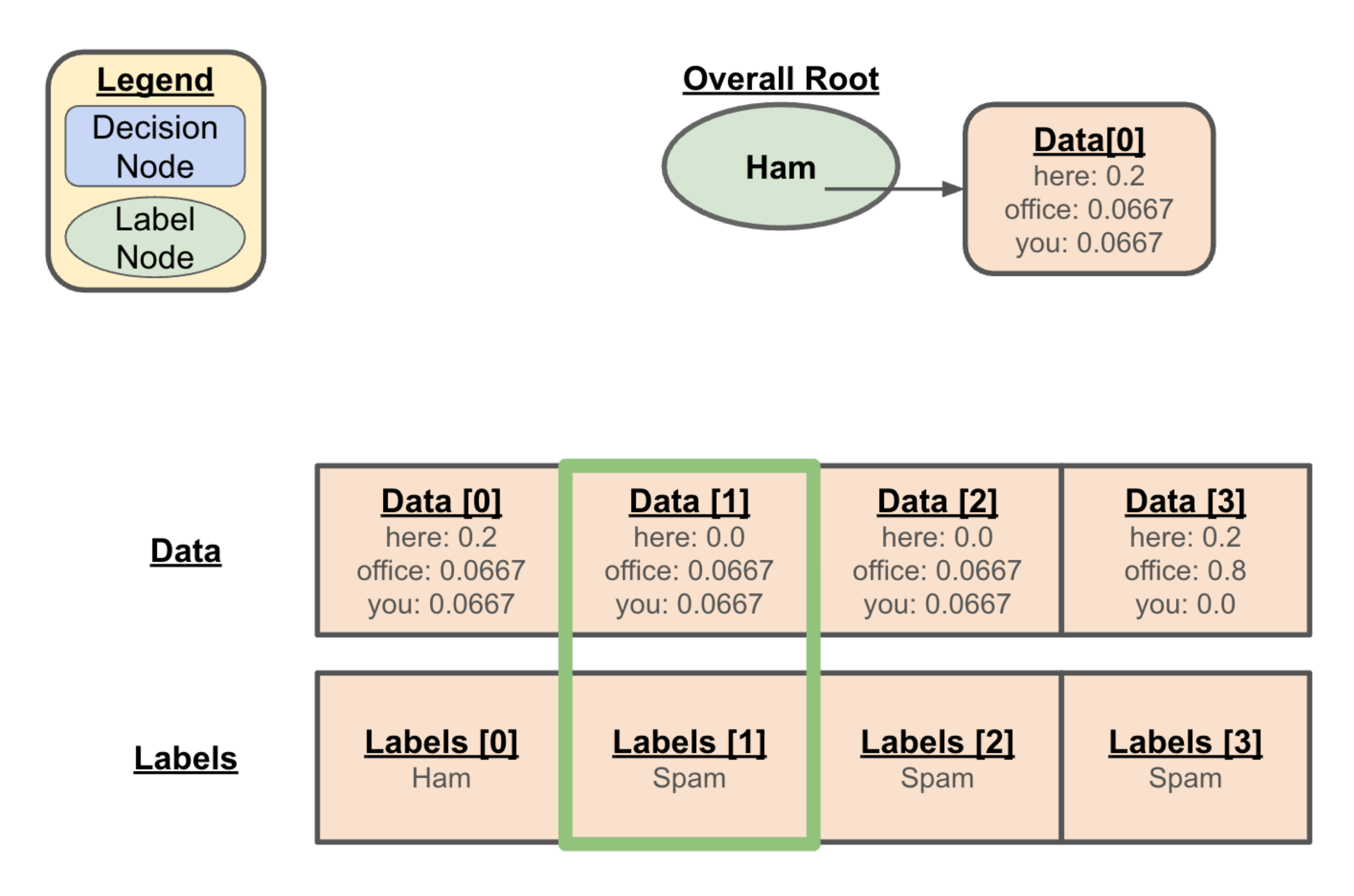
- If the prediction is correct, then our model is accurate up to that point, and we have nothing to do!
- If the prediction **is incorrect**, we need to **update the model** — this is the "learning" part. We modify the tree so that it can correctly classify this new example in the future.

Expand to see visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We process the next index, `1`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- Ham (root) (level 0) (stores Data[0])

Data List:

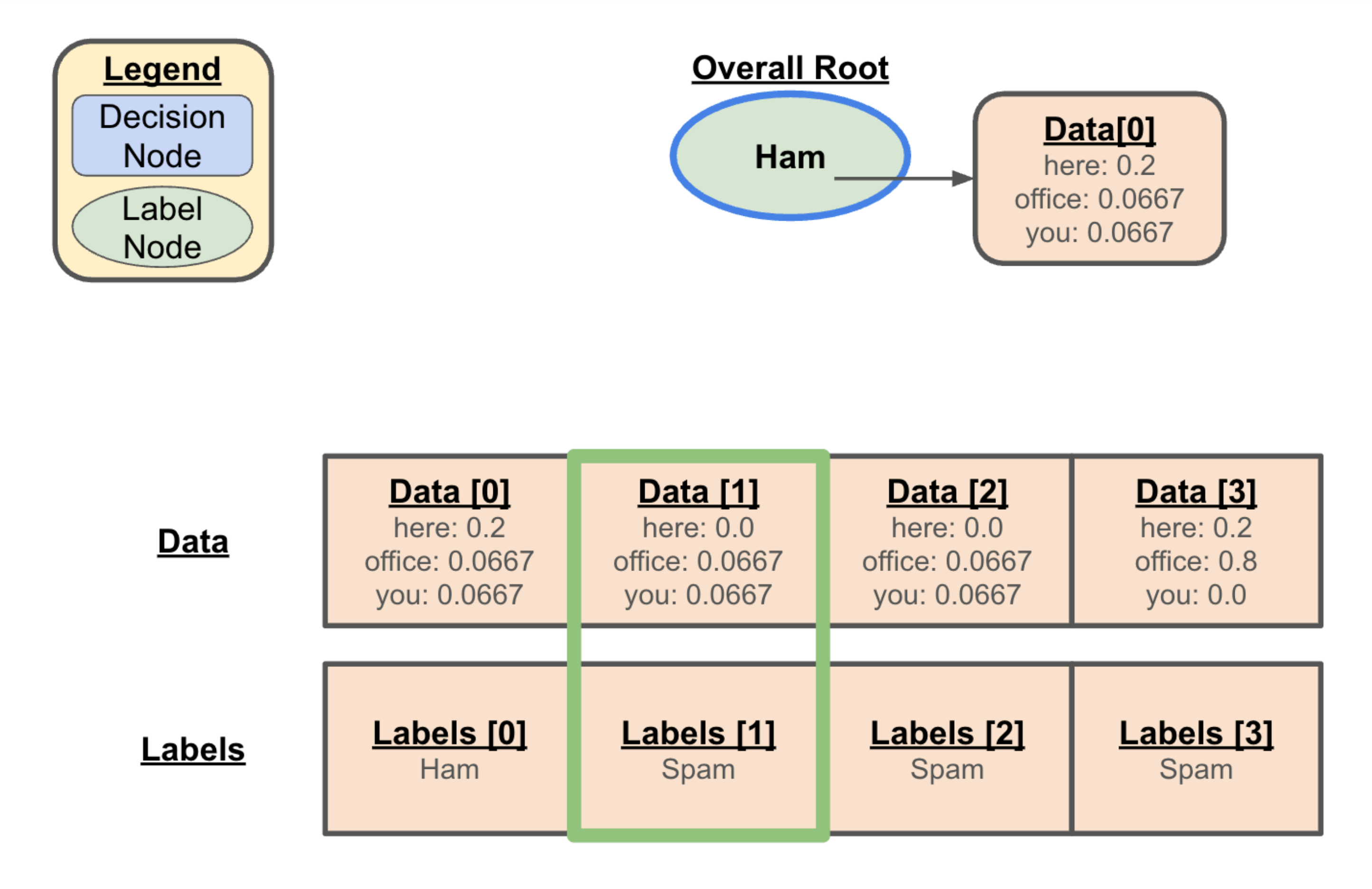
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0

    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

We classify the `TextBlock` from `data.get(1)`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0])

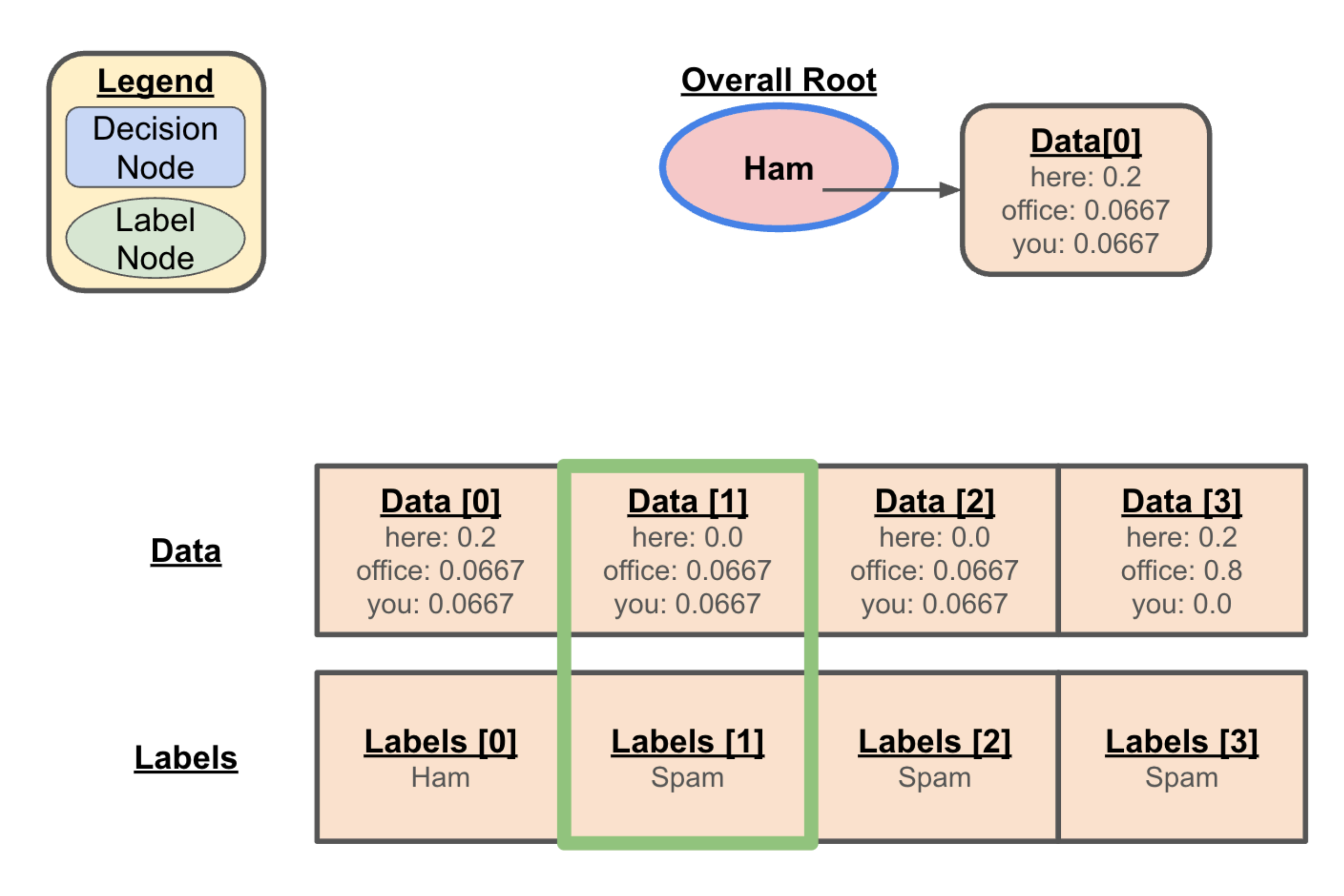
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Notice that the predicted label for `data.get(1)` (`Ham`) is different from our expected label of `Spam` (`labels.get(1)`). This indicates that we should update our model to adjust for this input.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0]) **incorrect label for the current list item**

Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:

  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Step 2a: Updating our Model

Typically, a large part of the complexity in building a classification tree is determining how to partition the data in case of incorrect predictions. There are many potential ways to accomplish this, but we'll be taking this approach:

- Call the `findBiggestDifference` method on the current `TextBlock` input and the previously stored one (from the misclassified label node).
  - `findBiggestDifference` identifies and returns the feature with the **largest difference in word probabilities** between the two `TextBlock`s.
- We then compute the **midpoint** between the two feature values in the `TextBlock`s and use it as the threshold for a new decision node.
- The decision node should be placed where the original label node currently is.
- After the new decision node is constructed, the original label node and the current input should be placed appropriately.

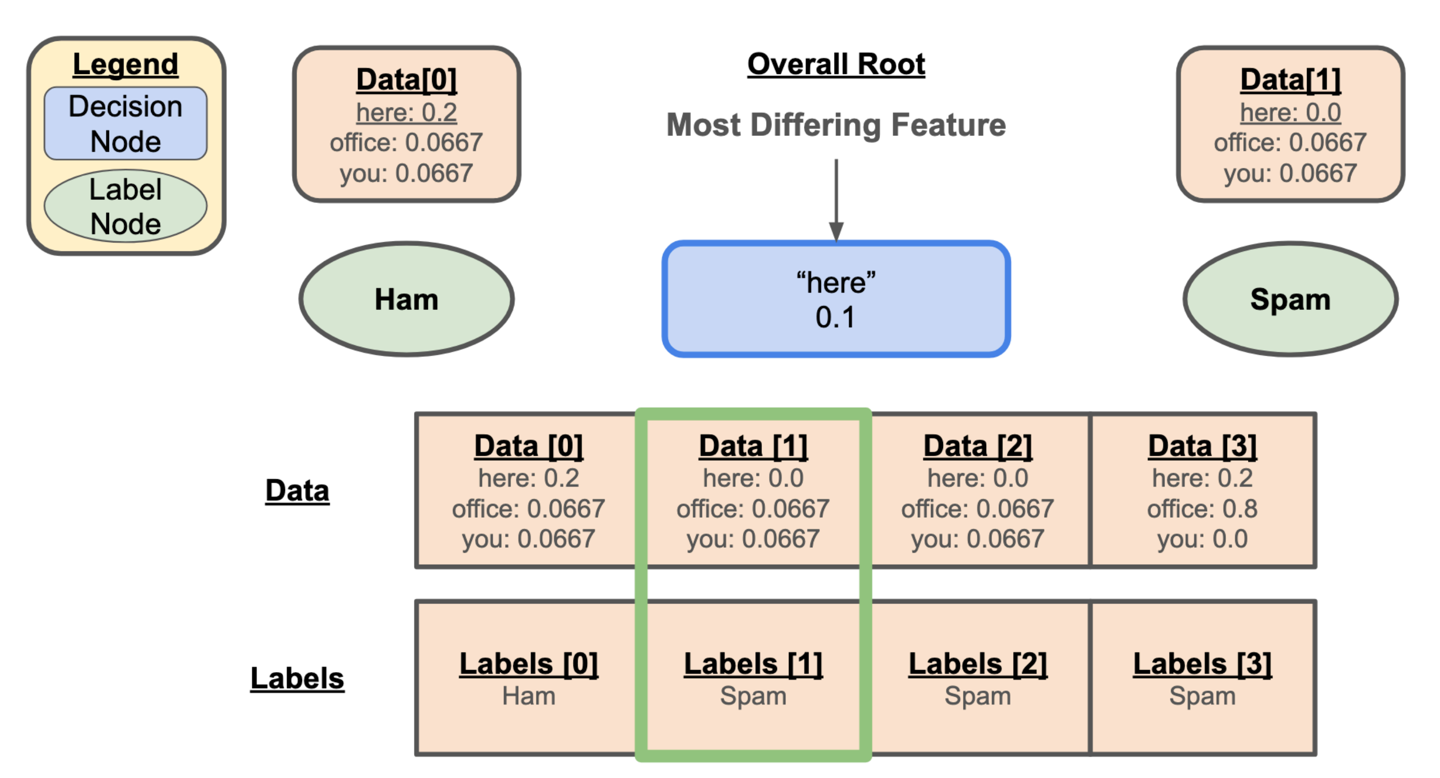
This step is why we needed to store the `TextBlock` along with its labels in the tree. Otherwise, without it, we would be unable to create a new feature for when our model is inaccurate!

i **NOTE:** We are only ever modifying the leaves of our tree!

Expand to see visualization:

▼ Expand

We find the most differing feature between the `TextBlock` we are currently processing (`data.get(1)`) and the `TextBlock` stored in the label node we were on (which in this case was from `data.get(0)`). From this, we create a new decision node with the most differing feature (`"here"`) and a threshold that is the midpoint of the two `TextBlock`s we are examining. For the feature `"here"`, the old `TextBlock` had a threshold of `0.2`, whereas the current `TextBlock` has a value of `0.0`. Thus, the threshold for our new node will be the midpoint of `0.2` and `0.0` which is `0.1`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Most Differing Feature:** "here" (difference value = 0.1)
- Ham (stores Data[0])
- Spam (stores Data[1])

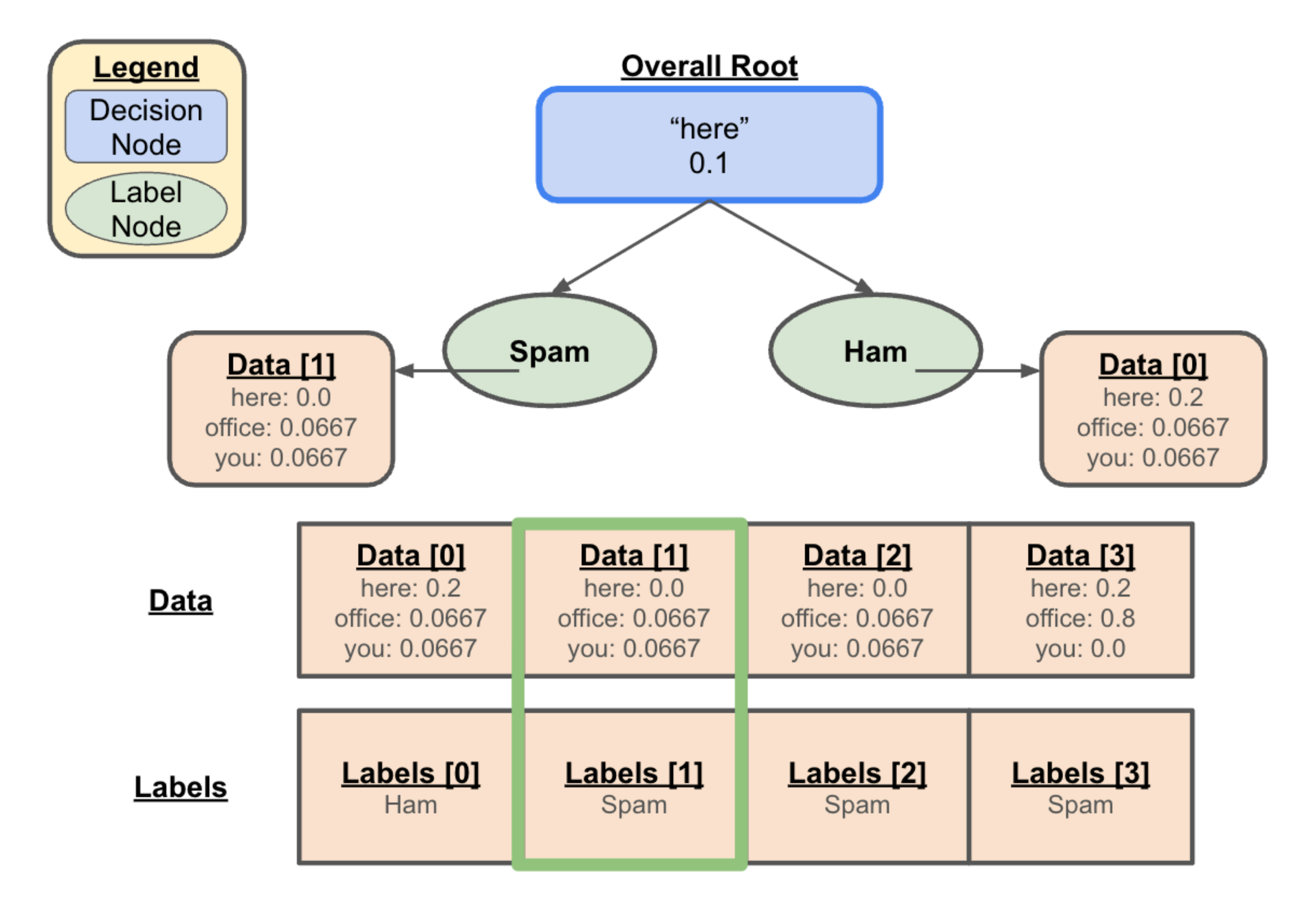
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item:** Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Notice that the label node with `data.get(1)` input is placed to the left of our new decision node because its `TextBlock` has a word frequency of `0` for `"here"` which is less than `0.1` On the other hand, notice that the label node with `data.get(0)` input is placed to the right of the new decision node because its `TextBlock` has a word frequency of `0.2` for `"here"`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item:** Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Now we've correctly updated our model to be able to correctly classify the data up to this point!

SIDE NOTE: This algorithm requires you to keep track of both the label and the `TextBlock` datapoint first assigned to this label within every leaf node created in this constructor, as without the previous `TextBlock` datapoint we would be unable to create a new decision node! Ideally we'd like to keep track of all input data that falls under a specific leaf node such that when creating a new decision node, 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.

Step 3: Repeat

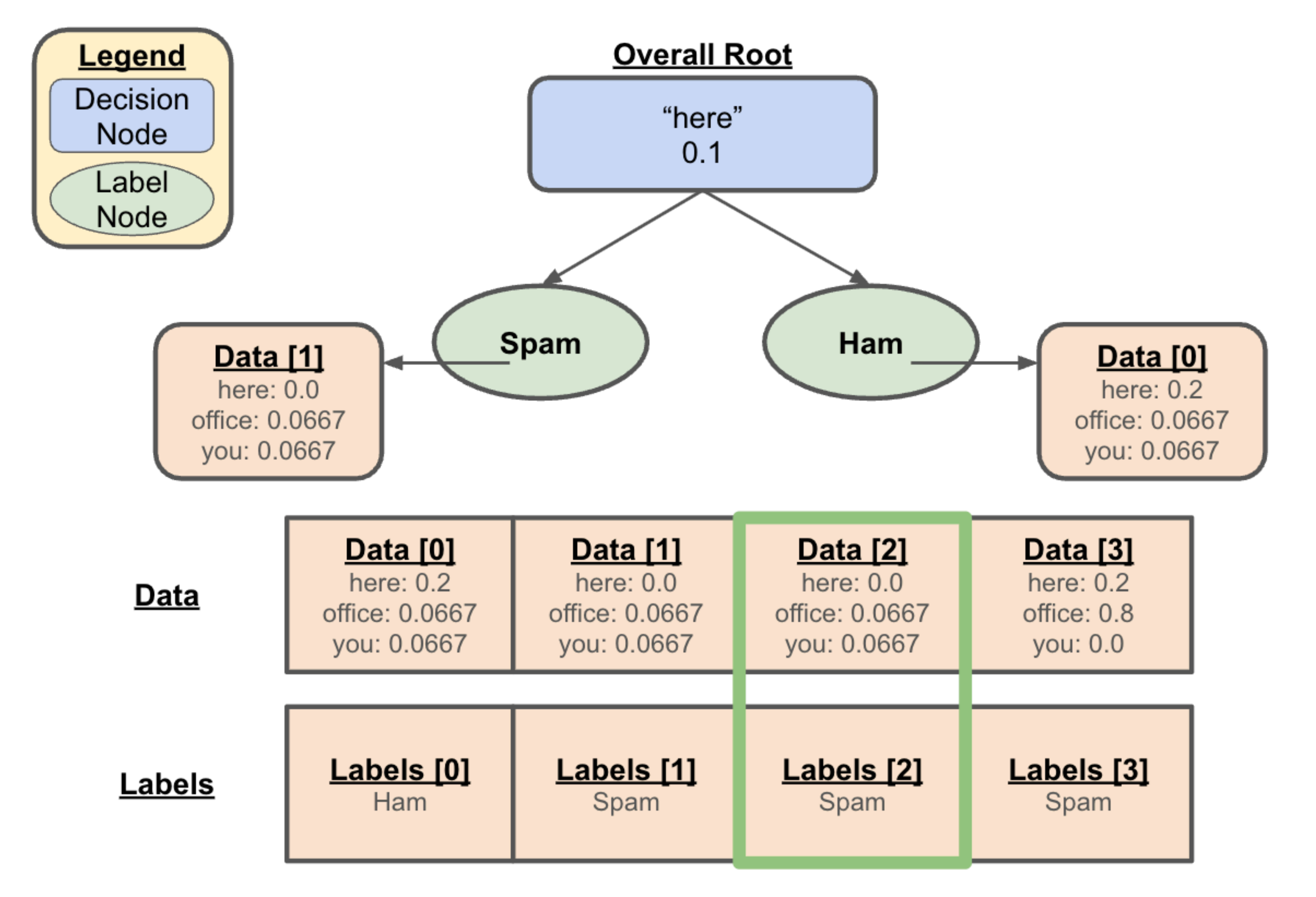
Repeat step 2 for the rest of the list until we've finished processing the list. At that point, our model is fully trained on our data and is able to predict the right label for the data we input.

Expand to see visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We're now processing index `2`. Start from the root of the tree and traverse through the existing tree until we reach a leaf node.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

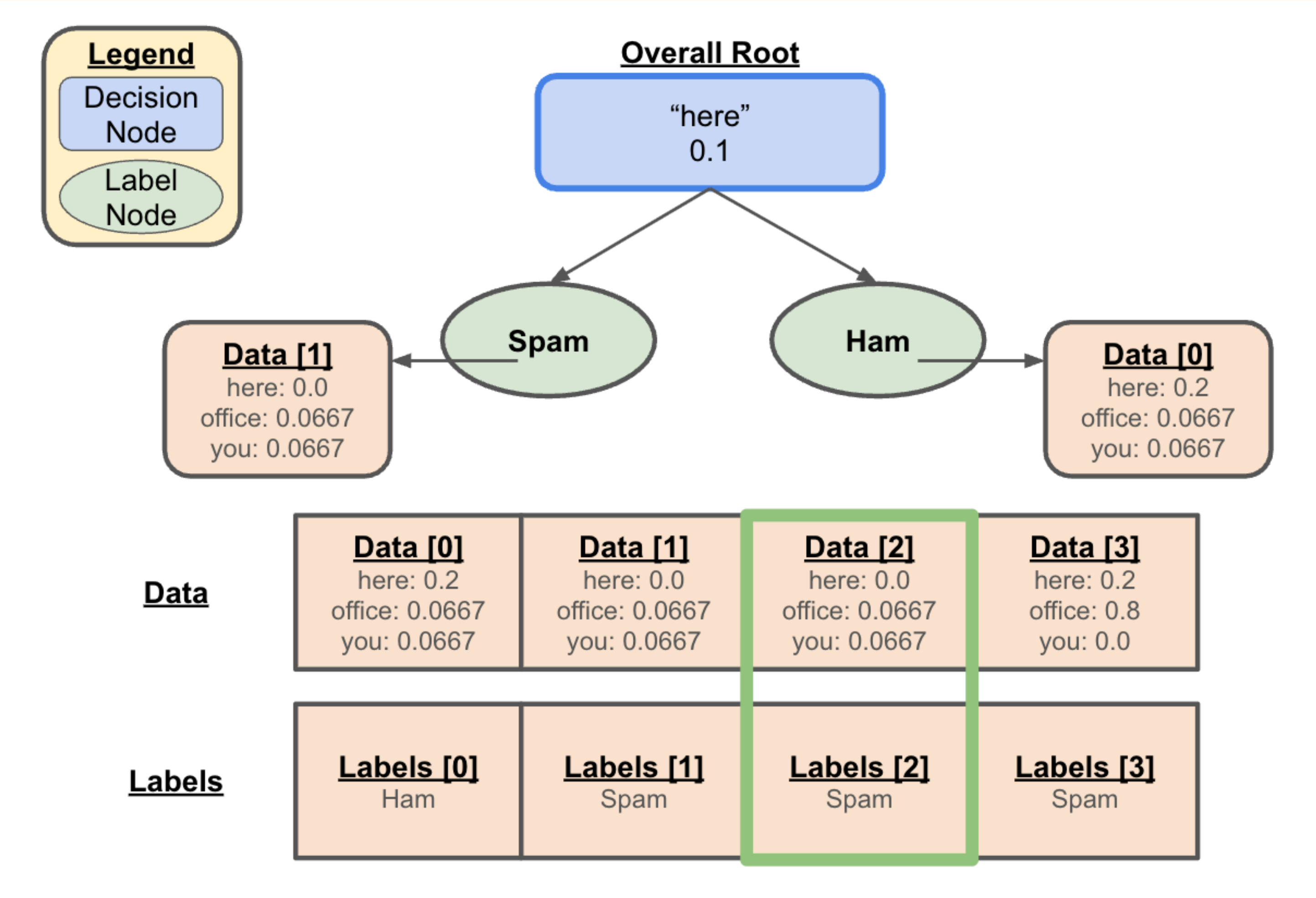
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:

    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

Since this datapoint's `"here"` probability is `0.0` which is less than `0.1`, we travel left:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1

- Spam (left) (level 1) (stores Data[1])
- Ham (right) (level 1) (stores Data[0])

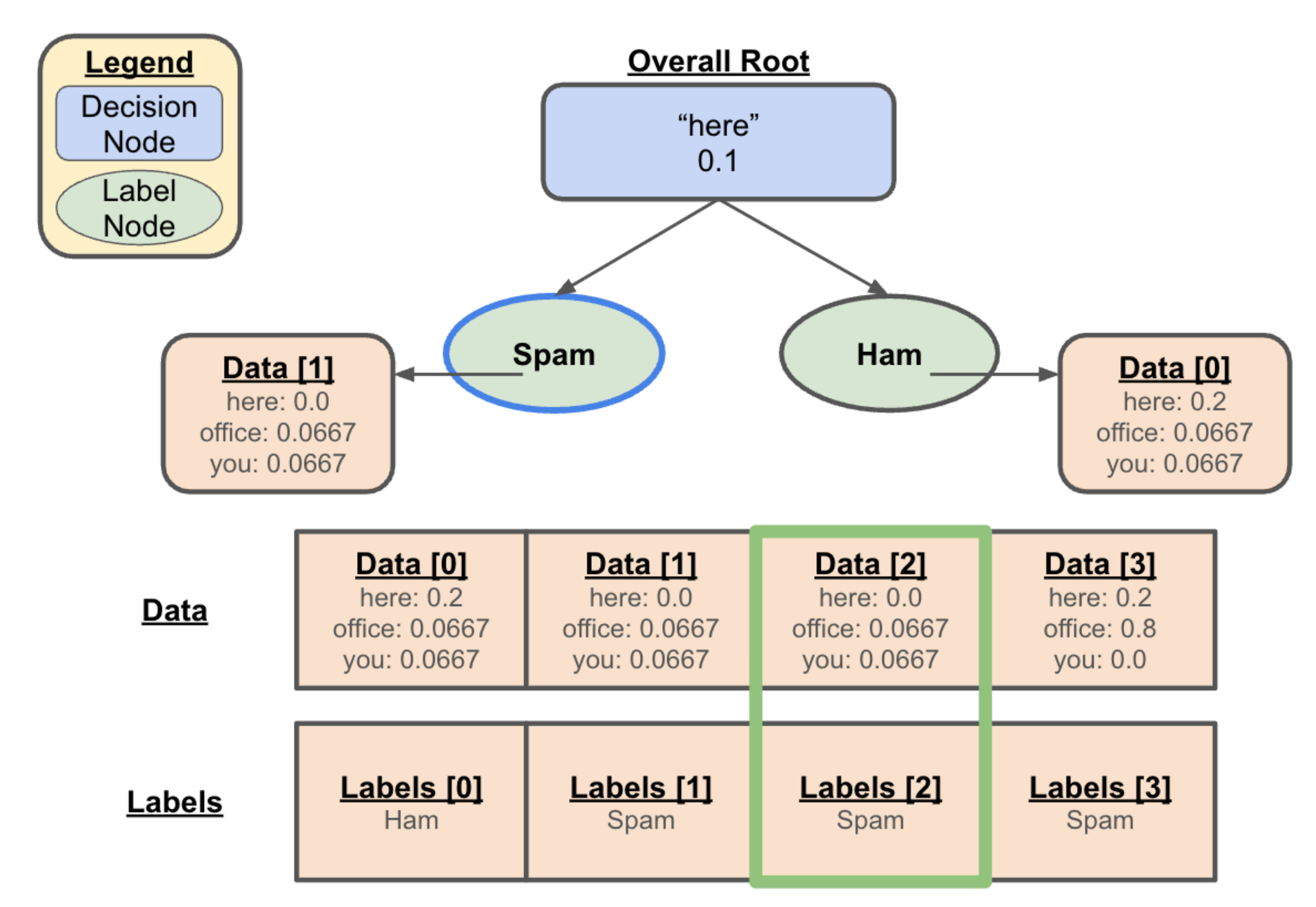
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

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!



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - **current node:** Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

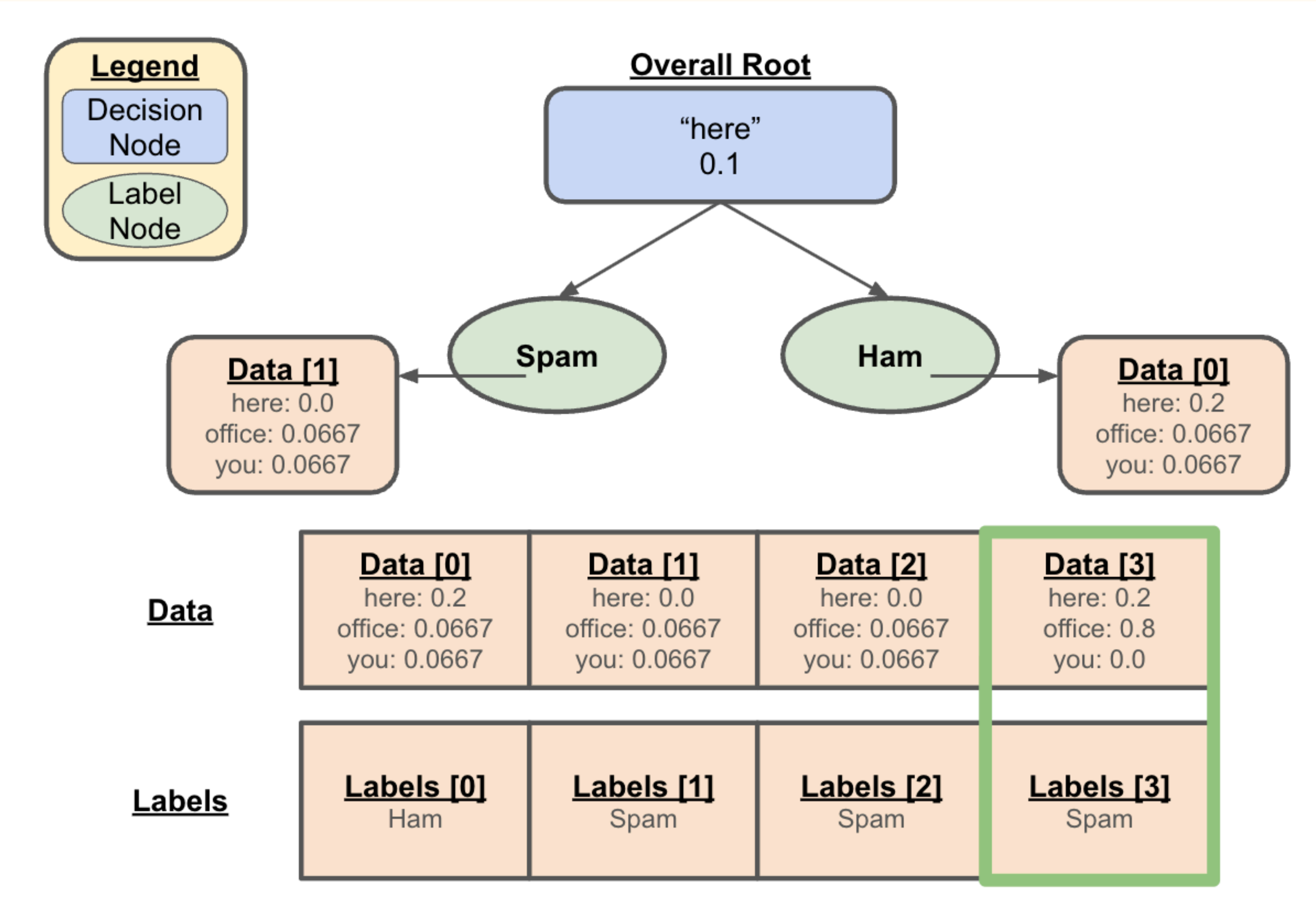
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

Lastly, we process index `3`. Start from the root of the tree and traverse through the existing tree until we reach a leaf node.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
    - Spam (left) (level 1) (stores Data[1])

- Ham (right) (level 1) (stores Data[0])

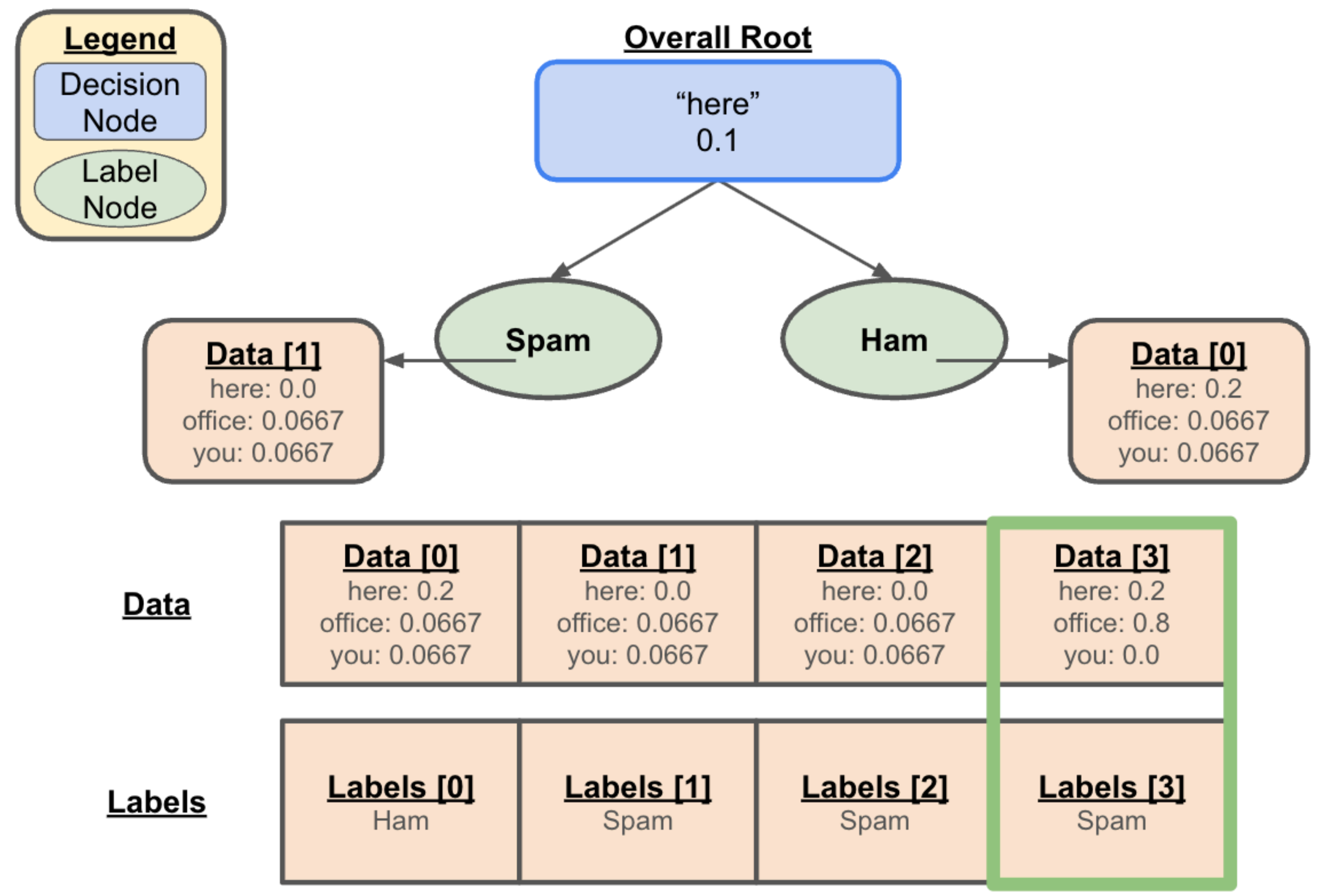
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

Since this datapoint's `"here"` probability is `0.2` which is greater than or equal to `0.1`, we travel right:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

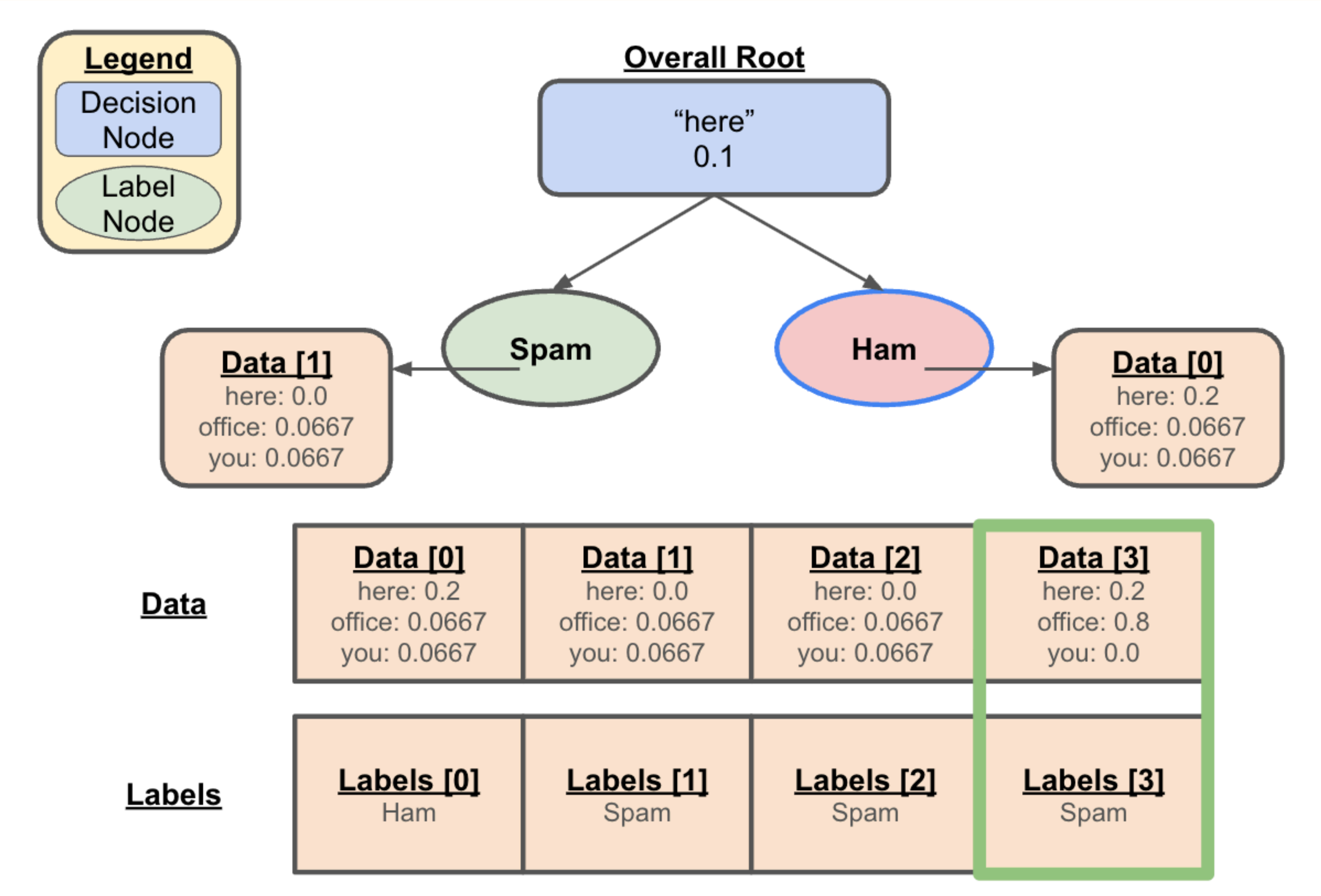
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

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 decision node with the most differing feature between the two `TextBlock` objects (one previously stored in the `Ham` node, and the other from `Data`) based on their `get()` values:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
    - Spam (left) (level 1) (stores Data[1])
    - **current node:** Ham (right) (level 1) (stores Data[0])

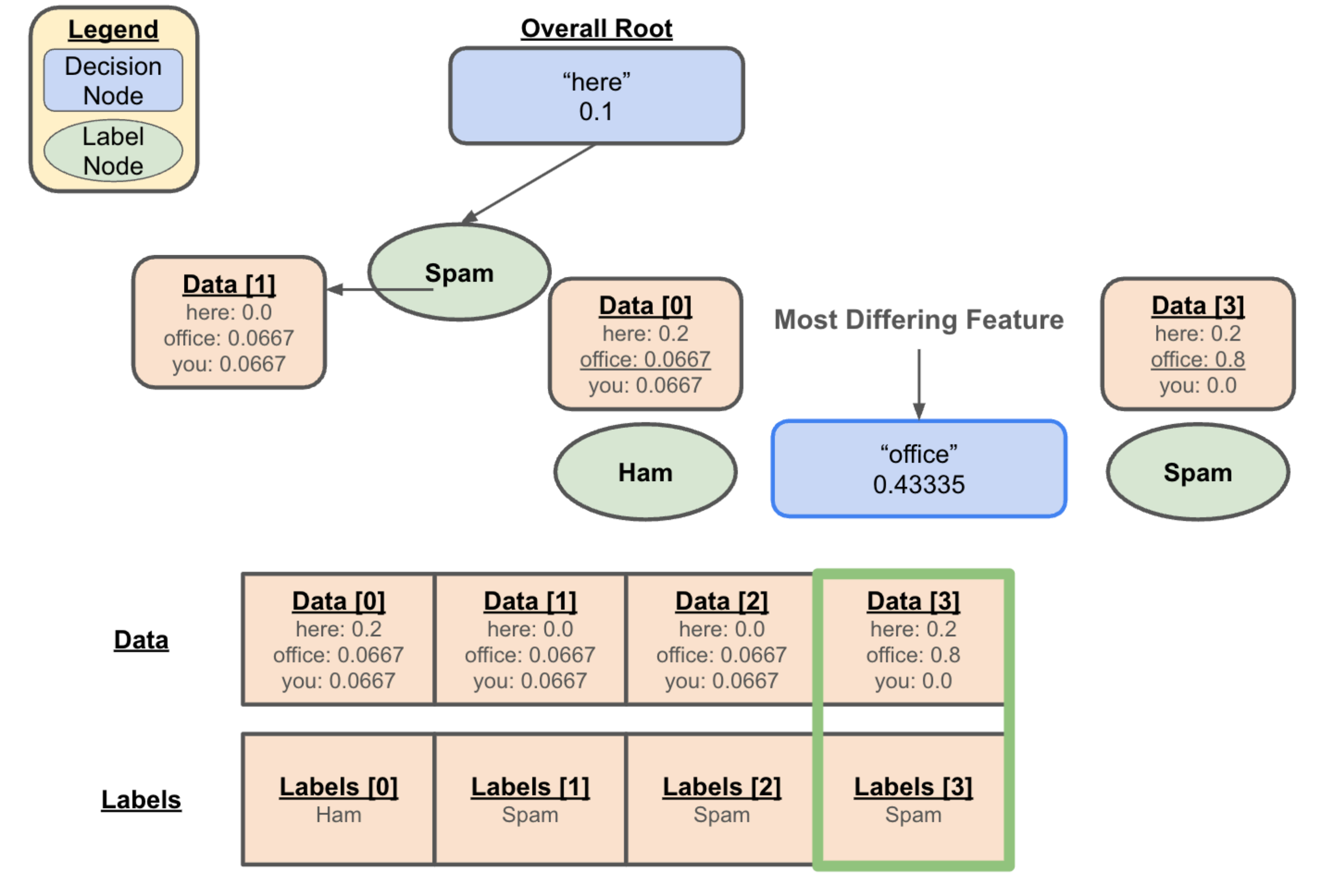
Data List:

- Data[0]:
    - word probability "here" = 0.2
    - word probability "office" = 0.0667
    - word probability "you" = 0.667
- Data[1]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- **Current item:** Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

We can then utilize the provided methods to produce a new decision node that will allow us to correctly distinguish `data.get(3)` vs. `data.get(0)` using a feature and a threshold based on the algorithm described in **Step 2a**. All that's left to do is organize the label nodes appropriately, as seen below:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

To The Right Of The Tree:

- Ham (stores Data[0])
- **current node:** "office" (middle) threshold = 0.43335
- Spam (stores Data[3])

Data List:

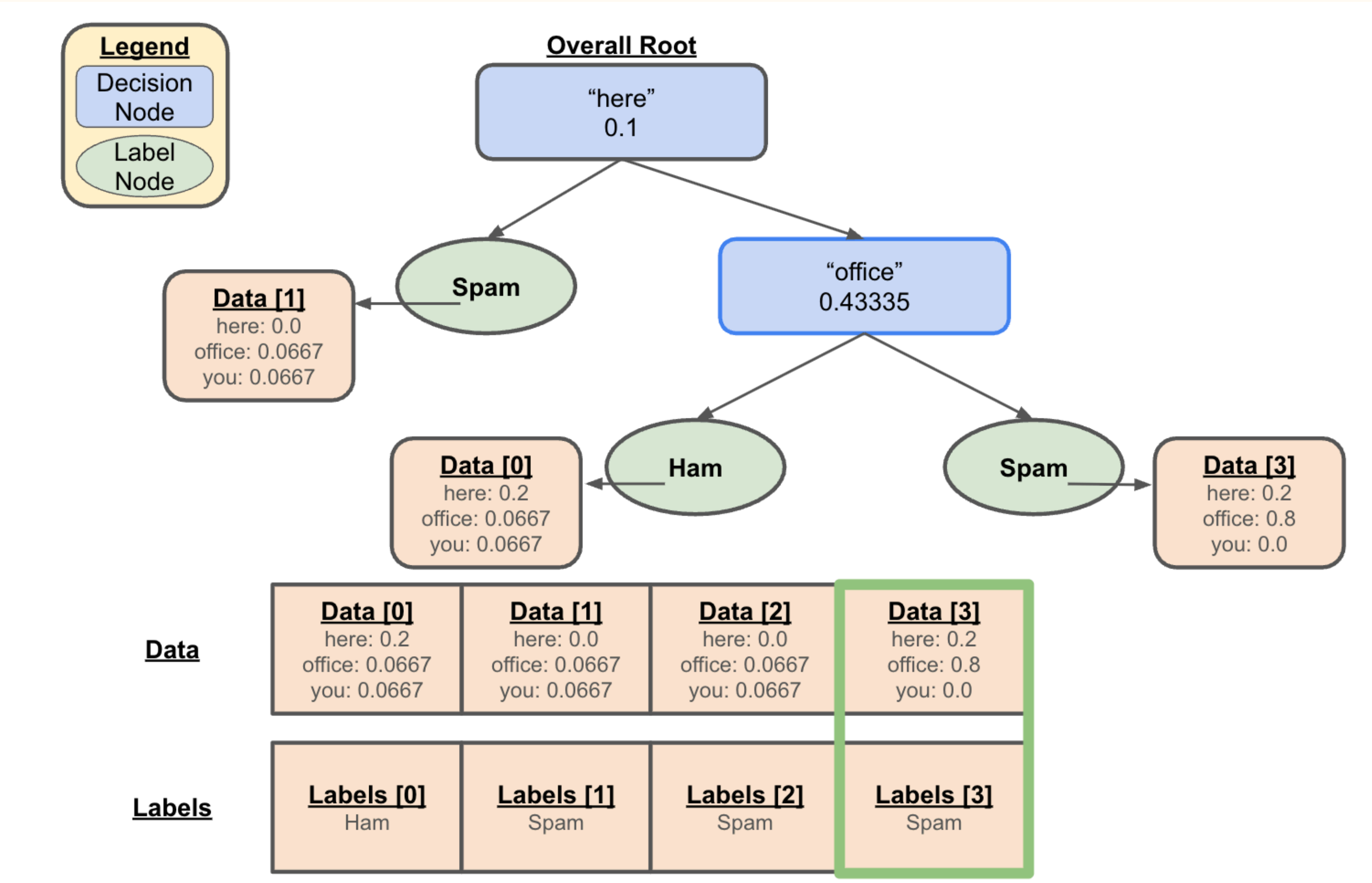
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0

- word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- **Current item:** Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

`data.get(0)` is placed to the left of our new decision node because it has a word frequency of `0.0667` for `"office"` which is less than `0.43335,` and `data.get(3)` is placed to the right of the new decision node because it has a word frequency of `0.8` for `"office"`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - **current node:** "office" (right) (level 1) decision node, threshold = 0.43335
    - Ham (left) (level 2) (stores Data[0])
    - Spam (right) (level 2) (stores Data[3])

Data List:

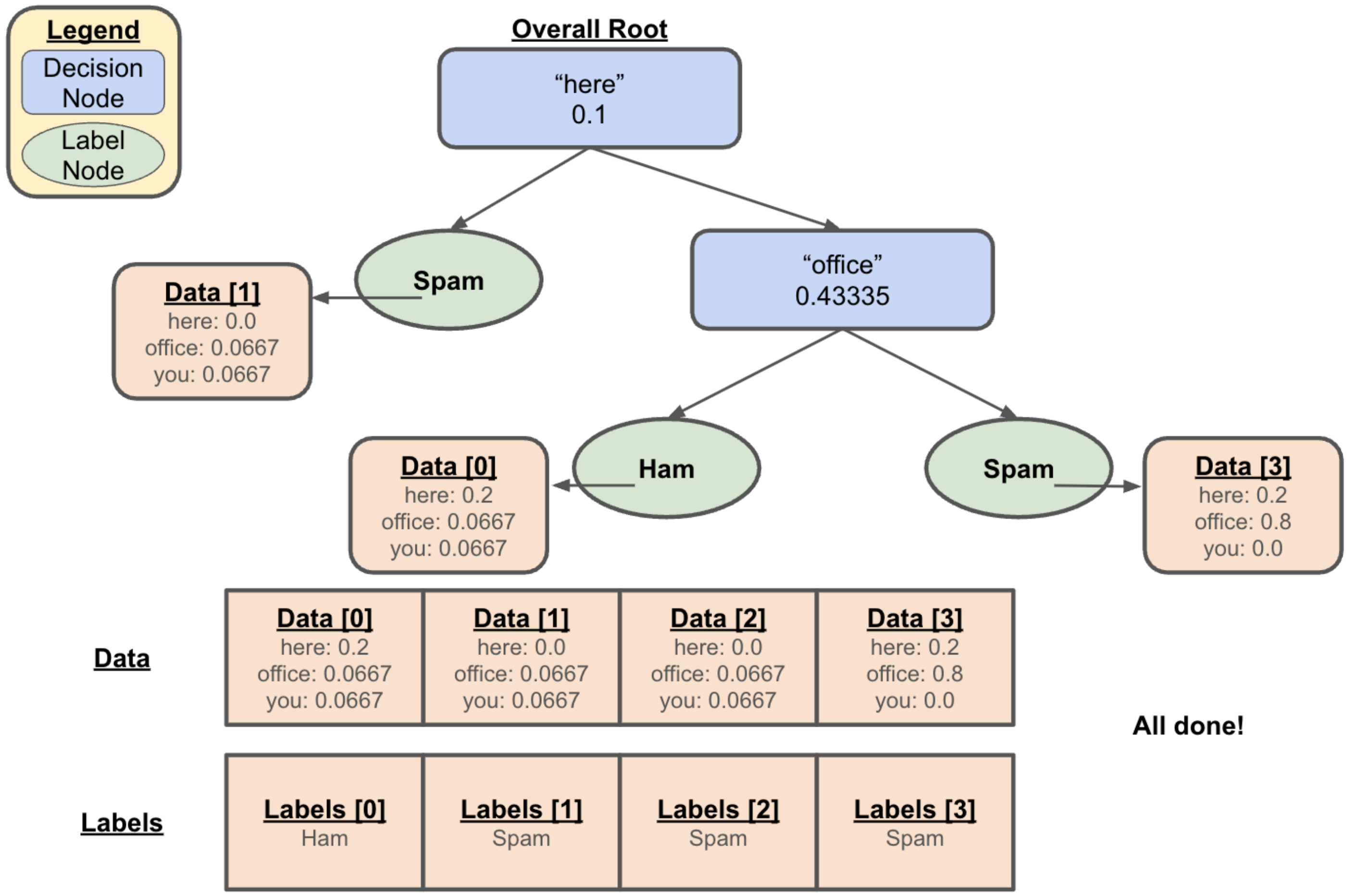
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

We've processed the entire list, so that means we're all done!

Expand to see an alternate equivalent representation of the above image:



▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - "office" (right) (level 1) decision node, threshold = 0.43335
    - Ham (left) (level 2) (stores Data[0])
    - Spam (right) (level 2) (stores Data[3])

Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

To The Right:

All done!

Implementation Requirements

Learning Objectives

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

- 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.

System Structure

Below, we describe the **provided** `TextBlock` class that will be used in your implementation of `Classifier.java` . You do not need to (and should not) make changes to this class, but your code will be a client of it. Make sure to understand the purpose of this class and read through the provided documentation.

TextBlock.java

Text data that gets classified via the classifier. It defines four public methods:

```
public double get(String word)
```

- Returns the corresponding word probability for the given word.
  - Although there are classification trees where it would make sense to work with other kinds of data that should return something else (imagine a color feature within a real estate dataset), since our implementation is only dealing with thresholds for word probabilities, this must return a double.

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

- Returns a set of all features from this text block.

```
public boolean containsFeature(String word)
```

- Returns true if this dataset contains the given word. False otherwise.

```
public String findBiggestDifference(TextBlock other)
```

- Returns the word with the biggest difference in probability between this `TextBlock` and the `other` `TextBlock`.
    - Note that there is no difference between calling `a.findBiggestDifference(b)` and `b.findBiggestDifference(a)`. Both will return the same string.

Required Class

> **NOTE:** To earn a grade higher than N on the Behavior and Concepts dimensions of this assignment, **your *core* algorithms for each method must be implemented *recursively*. You will want to utilize the *public-private pair* technique discussed in class.** You are free to create any helper methods you like, but the core of your implementations must be recursive.

Classifier.java

In this assignment, you implement your classification tree by creating a class called `Classifier.java`. You are provided with a client program that handles user interaction and calls your `Classifier` methods in order to train/load a model and classify data.

```
public Classifier(Scanner input)
```

- Load the classifier from a file connected to the given `Scanner`. The format of the input file should match that of the `save` method (described below).
    - Importantly, in this method, you should only read data from the file using `nextLine` and convert it to the appropriate format using `Double.parseDouble`.
- This method should throw an `IllegalArgumentException` if the provided `input` is null and an `IllegalStateException` if the tree is still empty after processing `input`.

```
public Classifier(List<TextBlock> data, List<String> labels)
```

- Create and train a classifier from the input data and corresponding labels.
- The lists should be processed in parallel in increasing order (i.e., process index 0, then 1, then 2, etc), where the label corresponding to `data.get(<index>)` can be found at `labels.get(<index>)`. The general construction process should be accomplished via the algorithm described in Training a Classification Tree slide.
- This method should throw an `IllegalArgumentException` if any of the following cases are met:
    - `data` or `labels` is null
    - `data` and `labels` are not the same size
    - `data` or `labels` is empty

> **HINT:** This algorithm requires you to keep track of the initial `TextBlock` used to create the label node. Without this initial `TextBlock`, we would be unable to create a new feature for when our model is inaccurate! Keeping this in mind, what may be one of the fields needed in the `ClassifierNode` class?

```java
public String classify(TextBlock input)
```

- Given a piece of data, return the appropriate label that this classifier predicts.
    - This method should model the steps taken in the Background and Structure slide: at every feature, 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 parameter `input` is null, throw an `IllegalArgumentException`.

```java
public void save(PrintStream output);
```

- Saves this current classifier to the given `PrintStream`
    - For our classifier tree, **this format should be pre-order**. Every branch node will print two lines of data, one for feature preceded by "Feature: " and one for threshold preceded by "Threshold: ". 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.**
- If the parameter `output` is null, throw an `IllegalArgumentException`.

Provided Methods

Additionally, we have provided two other methods in `Classifier.java`:

```java
public Map<String, Double> calculateAccuracy(List<TextBlock> data, List<String> labels)
```

- 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.

```java
private static double midpoint(double one, double two)
```

- Helper method to calculate the midpoint between two doubles.
    - **HINT:** This should be used in the `Classifier(List<TextBlock>, List<String>)` constructor to calculate the midpoint!

`ClassifierNode`

As part of writing your `Classifier` class, you should also have a **private static inner class** called `ClassifierNode` 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 `ClassifierNode` class that can represent both features and labels — you should *not* create separate classes for the different types of nodes.
    - You may find that since we are representing both features and labels in the same node class, some fields may be unused at times. This is completely okay!
- The `ClassifierNode` class must not contain any constructors or methods that are not used by the `Classifier` class.
- The fields of the `ClassifierNode` class must be `public`.

- All data fields should be declared `final` as well. This does not include fields representing the children of a node.

NOTE: You may get a `variable ___ might not have been initialized` error, in which case, you will have to explicitly initialize the values for *all* `final` fields in your node class — even if you do not plan to utilize the value.

File Format

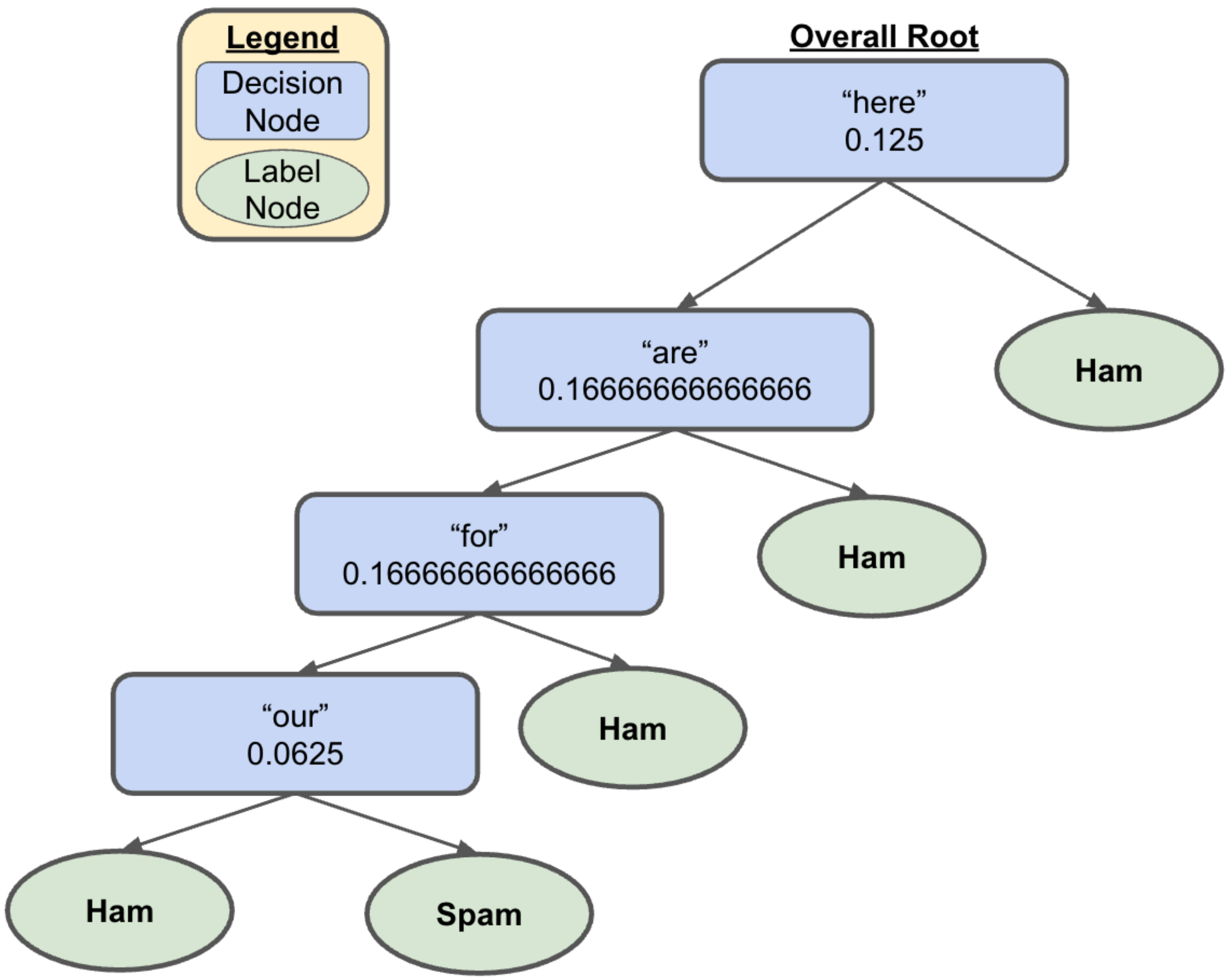
The files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent branch nodes and a single line to represent leaf nodes in the `Classifier`. The first line in each branch node pair will start 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: here
Threshold: 0.125
Feature: are
Threshold: 0.16666666666666666
Feature: for
Threshold: 0.16666666666666666
Feature: our
Threshold: 0.0625
Ham
Spam
Ham
Ham
Ham
```

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



Client Program & Visualization

We have provided you with a `Client` program to help test your implementation of the methods within `Classifier.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**).

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` (in this example, we initialized it in line 8 of `Client.java` to `"data/emails/test.csv"`) and use the model to predict labels for data points in a given `.csv` file.

> **NOTE:** When saving 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!
(Remember to edit the TRAIN_FILE and TEST_FILE class constants if you want to change the files b

To begin, enter your desired mode of operation:

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

What would you like to do with your model?

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 2
Overall: 0.8637632607481853
ham: 0.9961365099806826
spam: 0.0

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 1
Please enter the path to the file you'd like to test
Example: "./data/emails/test.csv"
File path: ./data/emails/test.csv
Results: [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, ham, h
...
ham, 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 (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 4
```

2. This client visualization uses the `Classifier` 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 `Classifier(List<TextBlock>, List<String>)`, 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).

`TRAIN_FILE` and `TEST_FILE` were set to `"data/federalist_papers/train.csv"` and `"data/federalist_papers/test.csv"` respectively.

▼ Expand

```
_______  ___      _______  _______  _______  ___   _______  ___   _______  ______
|       ||   |    |   _   ||       ||       ||   | |       ||   | |       ||    _ |
|       ||   |    |  |_|  ||  _____||  _____||   | |    ___||   | |    ___||   | ||
|       ||   |    |       || |_____ | |_____ |   | |   |___ |   | |   |___ |   |_||_
|      _||   |___ |       ||_____  ||_____  ||   | |    ___||   | |    ___||    __  |
|     |_ |       ||   _   | _____| | _____| ||   | |   |    |   | |   |___ |   |  | |
|_______||_______||__| |__||_______||_______||___| |___|    |___| |_______||___|  |_|
Welcome to the CSE 123 Classifier!
(Remember to edit the TRAIN_FILE and TEST_FILE class constants if you want to change the files b

To begin, enter your desired mode of operation:

1) Train classification model (Two List Constructor)
2) Load model from file (Scanner Constructor)
Enter your choice here: 1

Would you like to shuffle the data?
1) Yes (Recommended for testing finalized models)
2) No  (Recommended for debugging models)
2

What would you like to do with your model?

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 2
Overall: 0.6923076923076923
MADISON: 0.6666666666666666
JAY: 1.0
HAMILTON: 0.6666666666666666

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 3
Would you like to save to a file or output the classification tree to console?
```

```
1) Save to a file
2) Output classification tree to console
1
Please enter the file name you'd like to save to: destinationFile.txt

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
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

Your program should exactly reproduce the format and general behavior demonstrated in the Ed tests. Cornbear's recommended approach is as follows:

1) Design your Node

First, design your node class that represents both the branch and leaf nodes within your classification tree. Think about the information these nodes will be required to store based on the specification. Remember that in our classification tree, branch nodes represent decisions and leaf nodes represent classification labels.

> **NOTE:** You may find that since we are representing both features and labels in the same node class, some fields may be unused at times. This is completely okay!

Additionally, consider this hint for the two-list constructor:

> **HINT from Cornbear:** This algorithm requires you to keep track of the initial `TextBlock` used to create the label node. Without this initial `TextBlock`, we would be unable to create a new feature for when our model is inaccurate! Keeping this in mind, what may be one of the fields needed in the `ClassifierNode` class?

2) Scanner Constructor

This constructor will be given a `Scanner` that contains data produced by `save()`. In other words, the input for this constructor is the output you produced with `save()` so that Cornbear can load the trees you make!

Remember that this file is stored in pre-order format, where the feature and threshold for decision nodes are stored on two lines within the file:

```
Feature: here
Threshold: 0.125
```

And labels are present without any additional formatting:

```
ham
```

You may assume that "Feature" and "Threshold" will never be labels within the input file.

Remember that you should only ever call `.nextLine()` on the provided Scanner. You might be tempted to call `nextLine()` to read the feature then `next()` and `nextDouble()` to read the threshold, but remember that mixing token-based reading and line-based reading is not so simple.

Assuming you are trying to retrieve the value of the threshold, here is an alternative that uses a method called `parseDouble` in the `Double` class that allows you to use `nextLine()`:

```
double threshold = Double.parseDouble(input.nextLine().substring("Threshold: ".length()));
```

Lastly, you should throw an `IllegalArgumentException` is `input` is null and an `IllegalStateException` if the tree is still empty after processing `input`.

HINT from Cornbear: It looks like we're processing lines and using that information to *modify* our tree. Keeping in mind our recently learned concept, **what pattern should we employ to help implement this constructor?**

The tests for your `Scanner` constructor implementation are tied to a working `save` implementation. This means that once you feel comfortable with your solution you should move onto the next part, and test for both implementations at the same time.

Relevant Problem:

- Section 13: Load Tree

3) save()

Once you've implemented the `Scanner` constructor, do the opposite! Namely, given an already constructed classification tree, save it to the provided `PrintStream` via a **pre-order traversal**. Here is the file format as copied from the Implementation Guidelines slide:

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 branch nodes and a single line to represent leaf nodes in the `Classifier`. The first line in each branch 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.

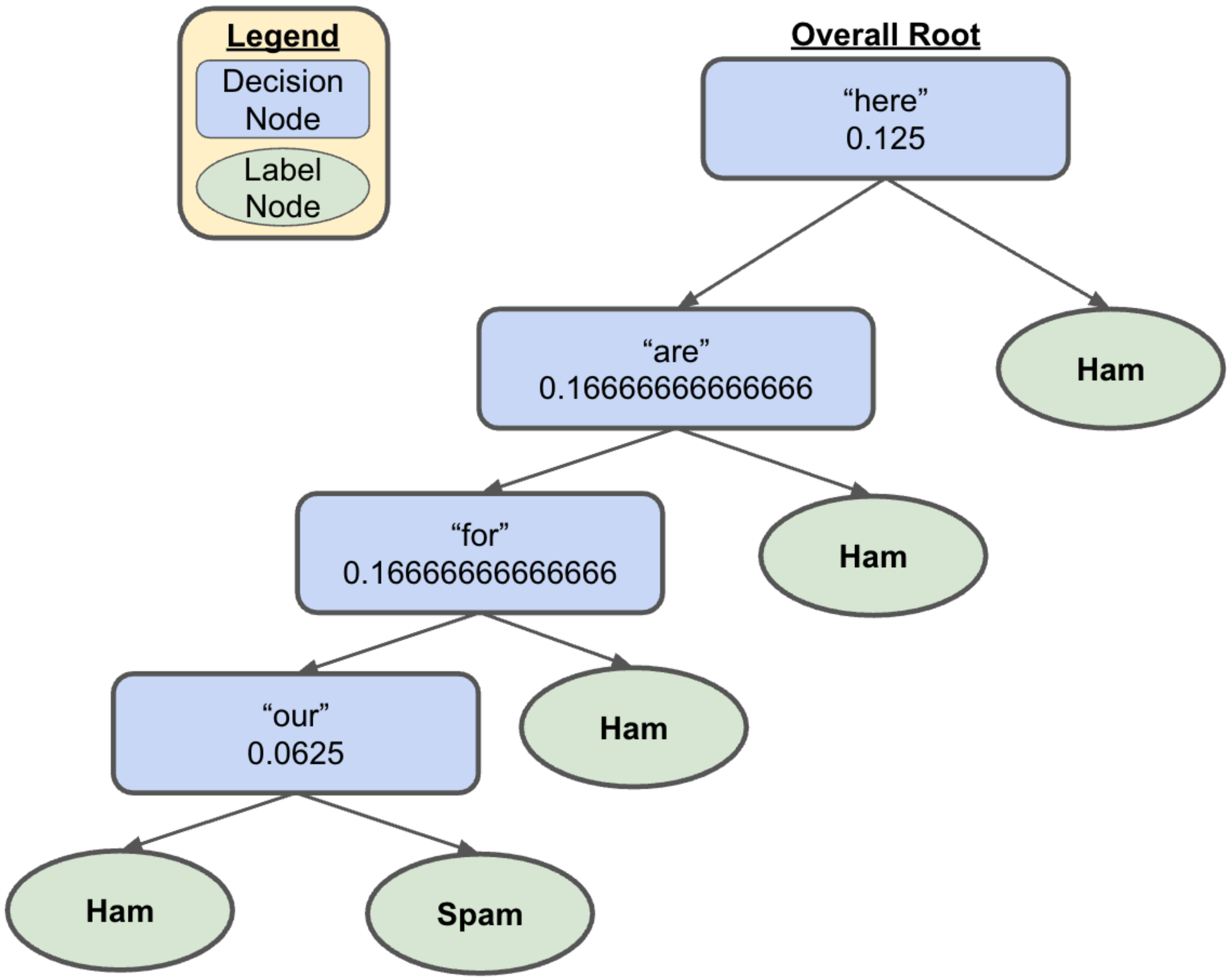
For example, consider the following sample file (`simple.txt`):

▼ Expand

```
Feature: here
Threshold: 0.125
Feature: are
Threshold: 0.16666666666666666
Feature: for
Threshold: 0.16666666666666666
Feature: our
Threshold: 0.0625
Ham
Spam
```

```
Ham
Ham
Ham
```

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



You should also throw an `IllegalArgumentException` is `output` is null.

> ⚠ At this point, test your `Scanner` constructor and `save` implementations. We don't recommend moving forward in this assignment until these two methods are passing the provided tests.

Below, we've provided sample client input and output that should be your expected output at this point (user input is **<u>bold and underlined</u>**):

▼ Expand

```
 _______  ___      _______  _______  _______  ___   _______  ___   _______  ______
|       ||   |    |   _   ||       ||       ||   | |       ||   | |       ||    _ |
|       ||   |    |  |_|  ||  _____||  _____||   | |    ___||   | |    ___||   | ||
|       ||   |    |       || |_____ | |_____ |   | |   |___ |   | |   |___ |   |_||_
```

```
|      _||   |___ |       ||_____  ||_____  ||   | |    ___||   | |    ___||    __  |
|     |_ |       ||   _   | _____| | _____| ||   | |   |    |   | |   |___ |   |  | |
|_______||_______||__| |__||_______||_______||___| |___|    |___| |_______||___|  |_|
Welcome to the CSE 123 Classifier!
(Remember to edit the TRAIN_FILE and TEST_FILE class constants if you want to change the files b

To begin, enter your desired mode of operation:

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

What would you like to do with your model?

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 3
Would you like to save to a file or output the classification tree to console?
1) Save to a file
2) Output classification tree to console
2

Save output:

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

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 4
```

Relevant Problems:

- Lesson 10: `printPreOrder`

- Section 13: Save Tree

4) classify()

Now we can start classifying! This method should traverse through the tree by evaluating decision nodes on the input data to see whether or not the input falls below the current threshold. If so, the traversal should continue into the left subtree; otherwise, the right. Once a leaf node is reached, the corresponding label should be returned.

For a feature at a given decision node, think about how we could retrieve its word probability from the input data.

Finally, you should throw an `IllegalArgumentException` is `input` is null

> ⚠ At this point, test your current implementation.We don't recommend moving forward runtil the `classify` method is passing

> ℹ **NOTE:** The `classify` tests are another way for us to test that your `Scanner` constructor implementation is correct (since testing using `save` alone doesn't guarantee its correctness). If you find your `classify` tests failing, but believe your `classify` implementation is correct, try taking a look at the logic inside your `Scanner` constructor!

Below, we've provided sample client input and output that should be your expected output at this point (user input is **<u>bold and underlined</u>**):

▼ Expand

```
 _______  ___      _______  _______  _______  ___   _______  ___   _______  ______
|       ||   |    |   _   ||       ||       ||   | |       ||   | |       ||    _ |
|       ||   |    |  |_|  ||  _____||  _____||   | |    ___||   | |    ___||   | ||
|       ||   |    |       || |_____ | |_____ |   | |   |___ |   | |   |___ |   |_||_
|      _||   |___ |       ||_____  ||_____  ||   | |    ___||   | |    ___||    __  |
|     |_ |       ||   _   | _____| | _____| ||   | |   |    |   | |   |___ |   |  | |
|_______||_______||__| |__||_______||_______||___| |___|    |___| |_______||___|  |_|
Welcome to the CSE 123 Classifier!
(Remember to edit the TRAIN_FILE and TEST_FILE class constants if you want to change the files b

To begin, enter your desired mode of operation:

1) Train classification model (Two List Constructor)
2) Load model from file (Scanner Constructor)
Enter your choice here: 2
Please enter the path to the file you'd like to load
Example: "./trees/simple.txt"
File path: ./trees/medium.txt

What would you like to do with your model?

1) Test with an input file (classify)
```

```
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 1
Please enter the path to the file you'd like to test
Example: "./data/emails/test.csv"
File path: ./data/emails/example_hams.csv
Results: [spam, ham, spam, spam, spam]

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 1
Please enter the path to the file you'd like to test
Example: "./data/emails/test.csv"
File path: ./data/emails/example_spams.csv
Results: [spam, spam, spam, spam, spam]

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 4
```

5) Two List Constructor

Finally, here is where we actually "train" our model, and this will likely be the most difficult part of your implementation. First, you should make sure to throw the proper exceptions:

- `IllegalArgumentException` if any of the following cases are met:
  - `data` or `labels` is null
  - `data` and `labels` are not the same size
  - `data` or `labels` is empty

Next, your implementation should follow the following algorithmic approach (copied from the Training a Classification Tree):

Step 1: Initialize our Model

Since the classification tree is empty at the beginning, we need to add an initial data point so it can start making classifications. This algorithm processes training examples in order, starting from index `0`. So we begin by inserting the first data-label pair (at index 0) into the tree.

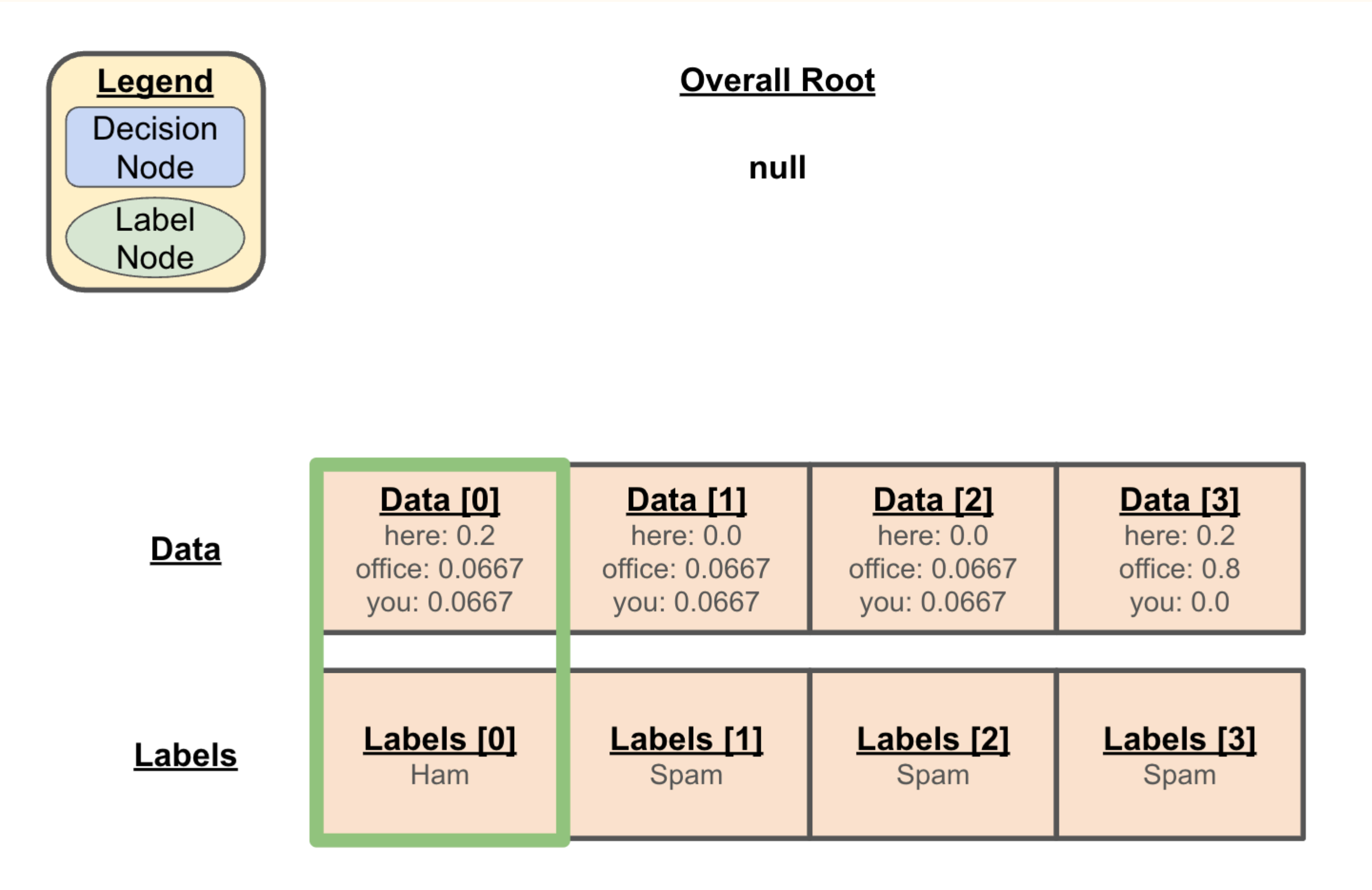
We also want to store the `TextBlock` data along with its label in the tree's leaves. This way, the classification tree can use previous examples to help make decisions when creating new nodes. If this part isn't entirely clear yet, that's okay. We will see this action later in the algorithm explanation.

Expand to see the visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We process index `0`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- null (root)

Data List:

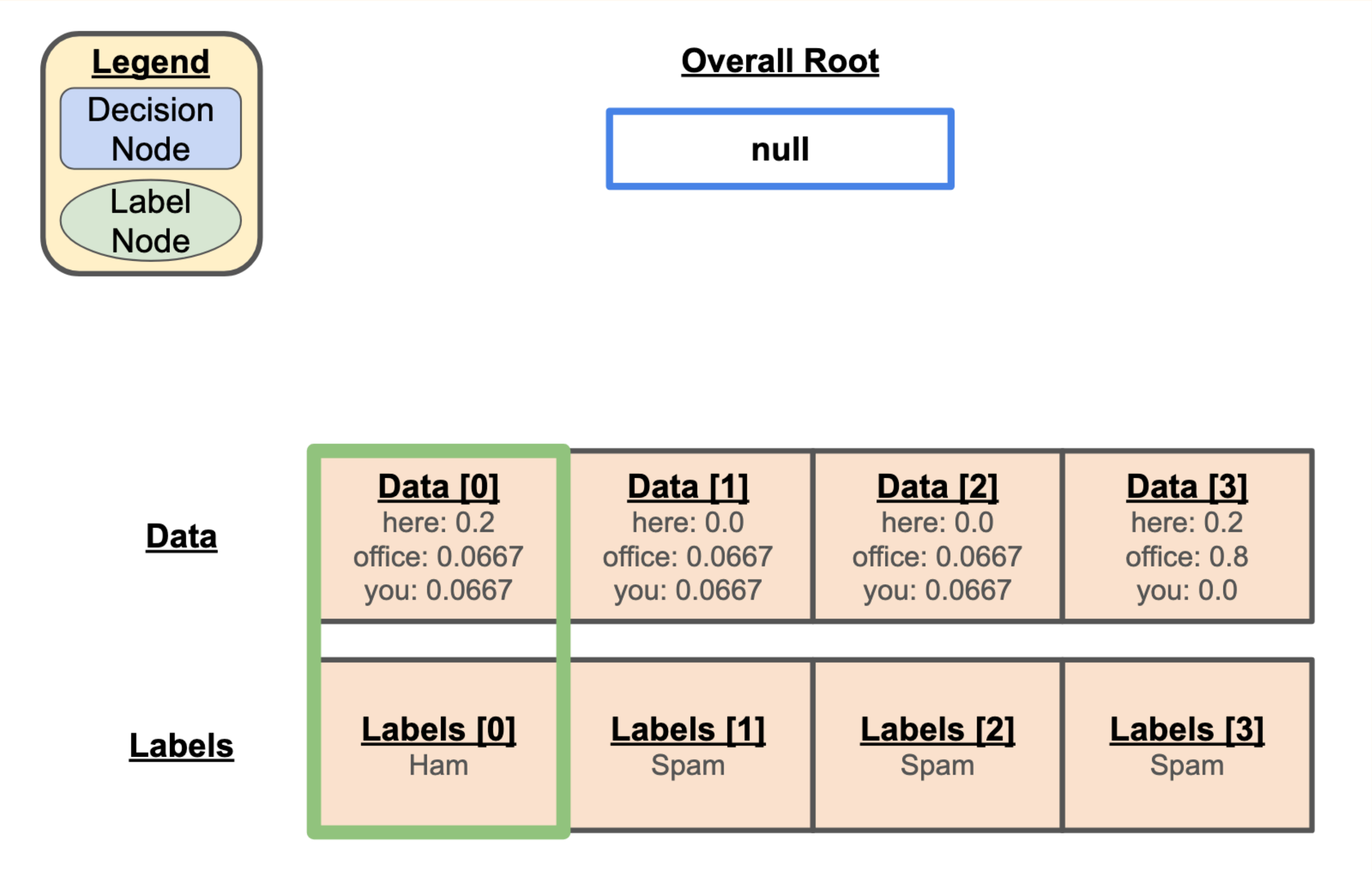
- **Current item:** Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0

- word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

The current tree is empty.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** null (root)

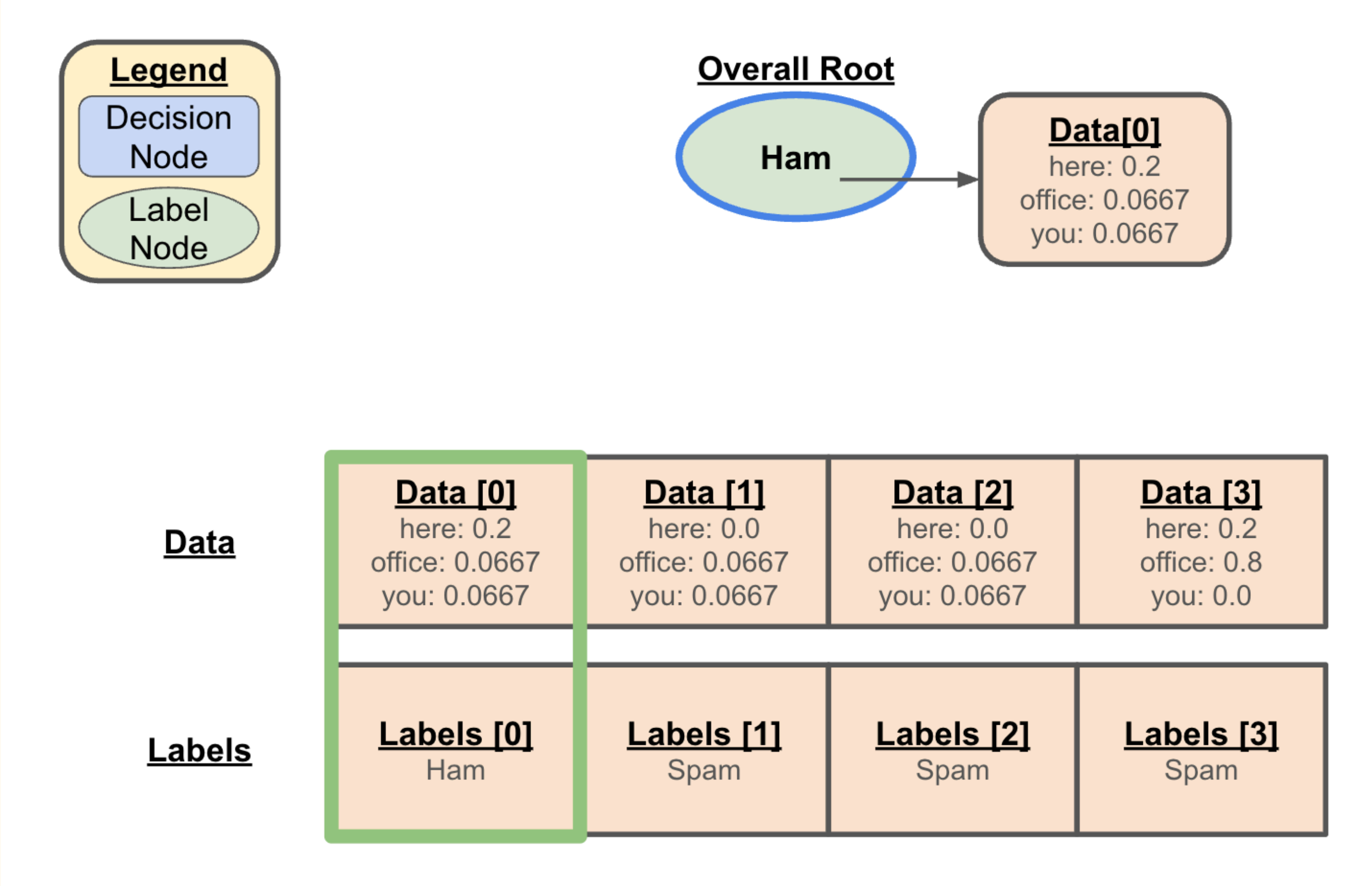
Data List:

- **Current item:** Data[0]:
    - word probability "here" = 0.2
    - word probability "office" = 0.0667
    - word probability "you" = 0.667
- Data[1]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

We create a new label node to store the information at index `0` since the tree is currently empty.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0])

Data List:

- **Current item:** Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8

- word probability "you" = 0.0

Labels List:

- **Current item:** Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Step 2: Classify Data

Now that we have a classification tree, we can start to classify inputs! Unfortunately, with only one point of data, our model doesn't seem very useful — currently, it classifies every input as `Ham`. What if we try to classify a piece of data that has an expected label of `Spam`?

To handle this, we proceed to the next step of the algorithm: we process the next index. We'll **start at the top of the tree** and traverse down to find the label our model will predict for `data.get(index)`. Now, we check whether our model's prediction matches the expected label.

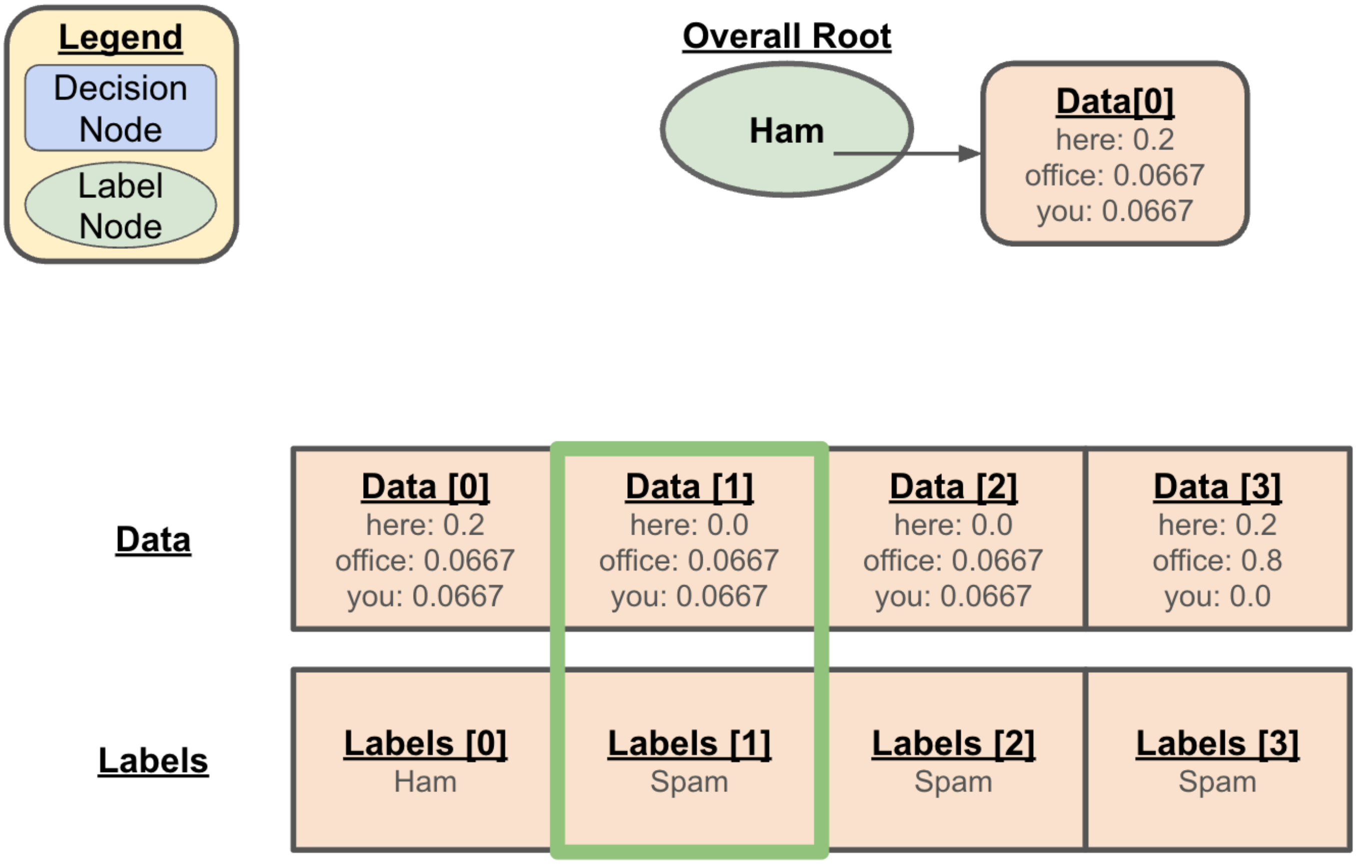
- If the prediction is correct, then our model is accurate up to that point, and we have nothing to do!
- If the prediction **is incorrect**, we need to **update the model** — this is the "learning" part. We modify the tree so that it can correctly classify this new example in the future.

Expand to see visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We process the next index, `1`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- Ham (root) (level 0) (stores Data[0])

Data List:

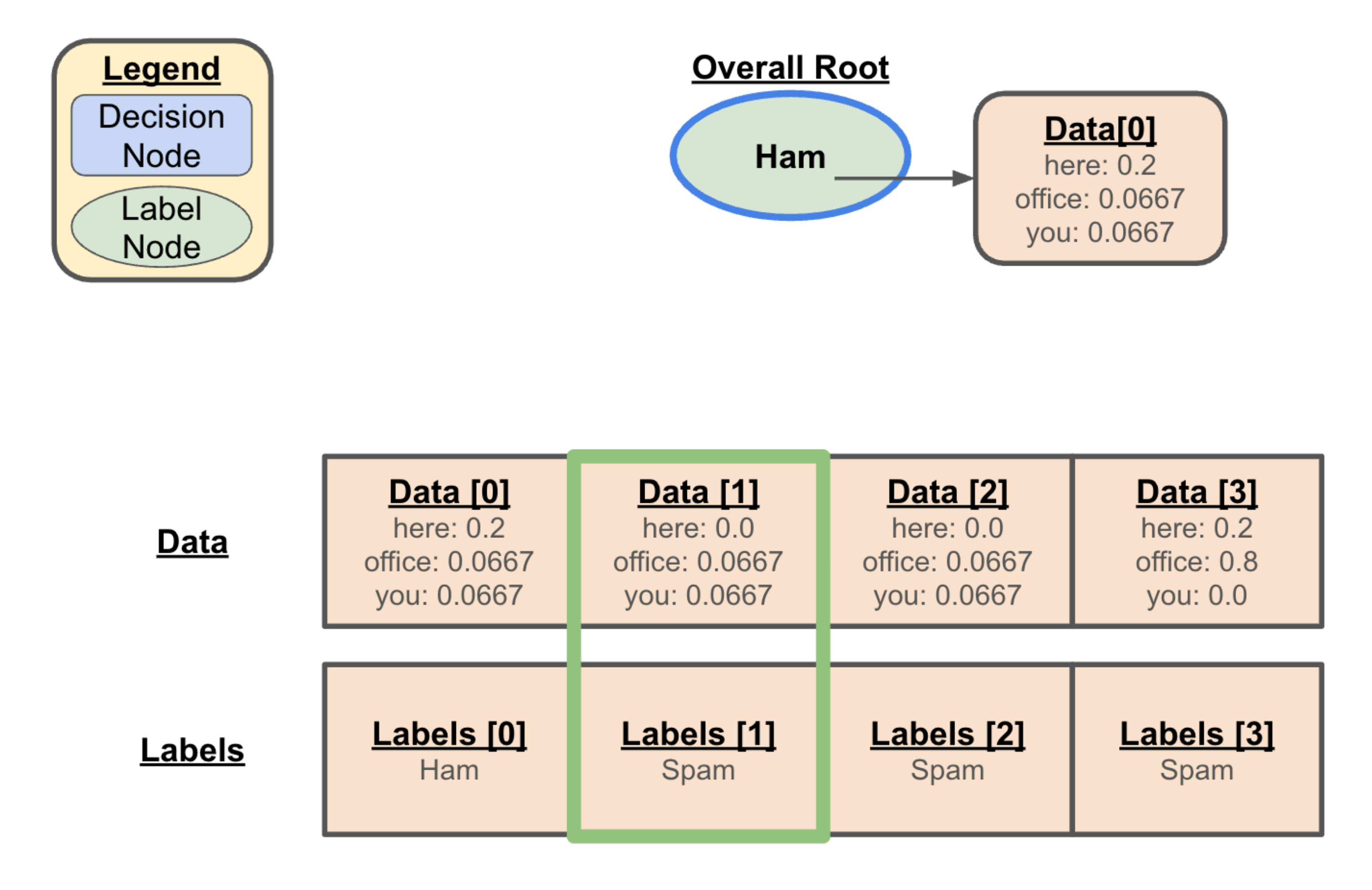
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2

- word probability "office" = 0.8
- word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

We classify the `TextBlock` from `data.get(1)`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0])

Data List:

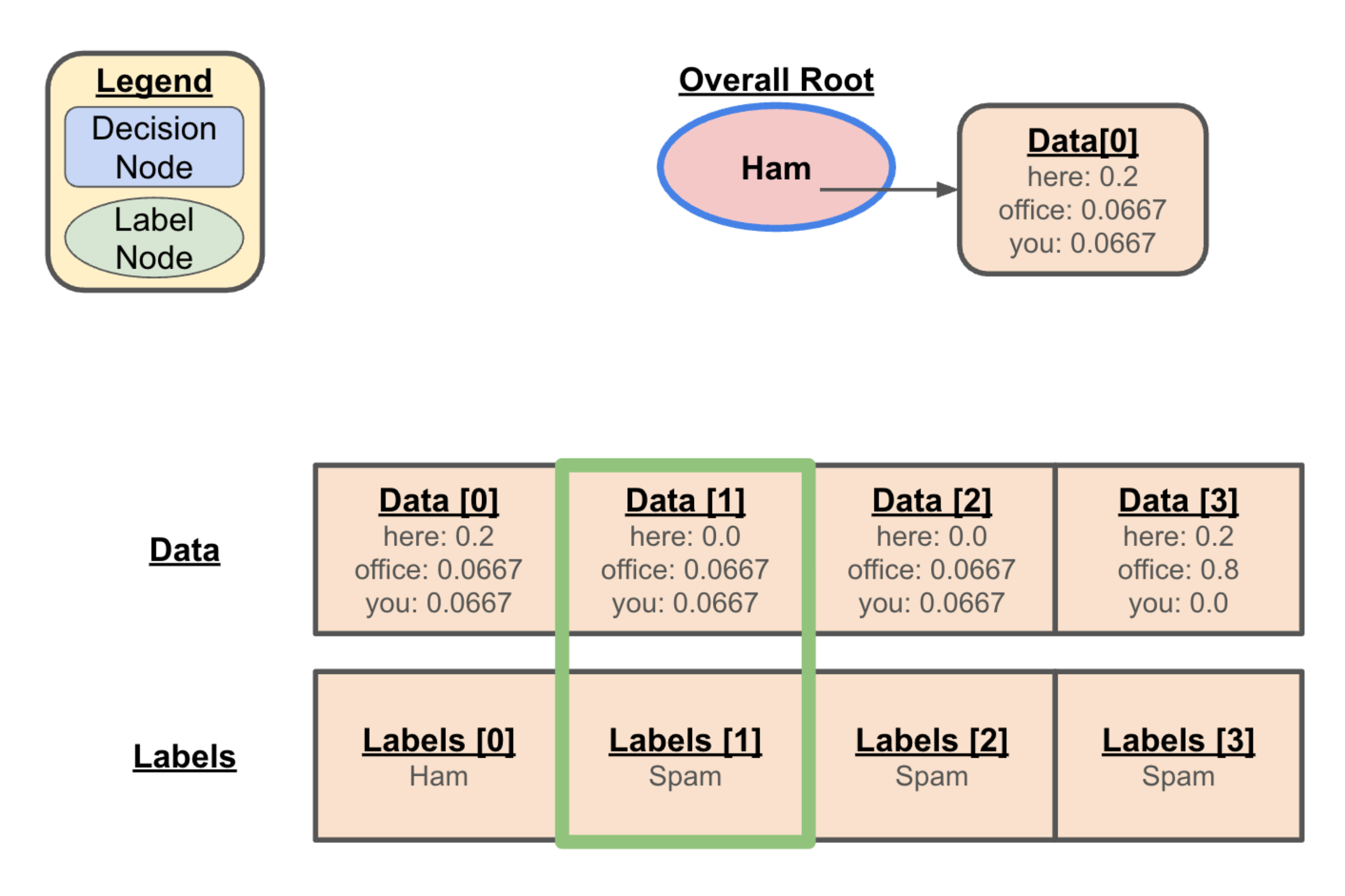
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667

- word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Notice that the predicted label for `data.get(1)` (`Ham`) is different from our expected label of `Spam` (`labels.get(1)`). This indicates that we should update our model to adjust for this input.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Current node:** Ham (root) (level 0) (stores Data[0]) **incorrect label for the current list item**

Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item**: Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Step 2a: Updating our Model

Typically, a large part of the complexity in building a classification tree is determining how to partition the data in case of incorrect predictions. There are many potential ways to accomplish this, but we'll be taking this approach:

- Call the `findBiggestDifference` method on the current `TextBlock` input and the previously stored one (from the misclassified label node).
  - `findBiggestDifference` identifies and returns the feature with the **largest difference in word probabilities** between the two `TextBlock`s.

- We then compute the **midpoint** between the two feature values in the `TextBlock`s and use it as the threshold for a new decision node.
- The decision node should be placed where the original label node currently is.
- After the new decision node is constructed, the original label node and the current input should be placed appropriately.

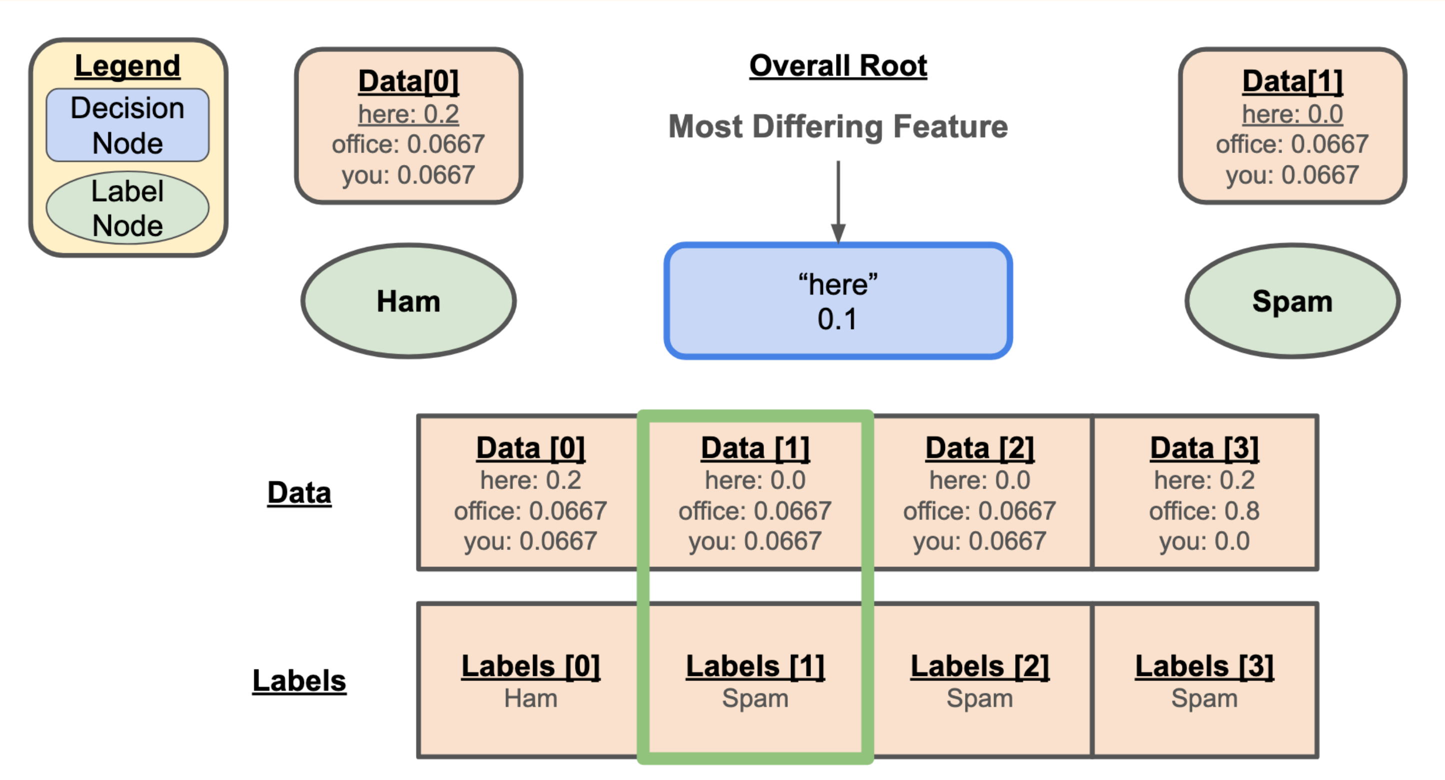
This step is why we needed to store the `TextBlock` along with its labels in the tree. Otherwise, without it, we would be unable to create a new feature for when our model is inaccurate!

NOTE: We are only ever modifying the leaves of our tree!

Expand to see visualization:

▼ Expand

We find the most differing feature between the `TextBlock` we are currently processing (`data.get(1)`) and the `TextBlock` stored in the label node we were on (which in this case was from `data.get(0)`). From this, we create a new decision node with the most differing feature (`"here"`) and a threshold that is the midpoint of the two `TextBlock`s we are examining. For the feature `"here"`, the old `TextBlock` had a threshold of `0.2`, whereas the current `TextBlock` has a value of `0.0`. Thus, the threshold for our new node will be the midpoint of `0.2` and `0.0` which is `0.1`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **Most Differing Feature:** "here" (difference value = 0.1)
- Ham (stores Data[0])
- Spam (stores Data[1])

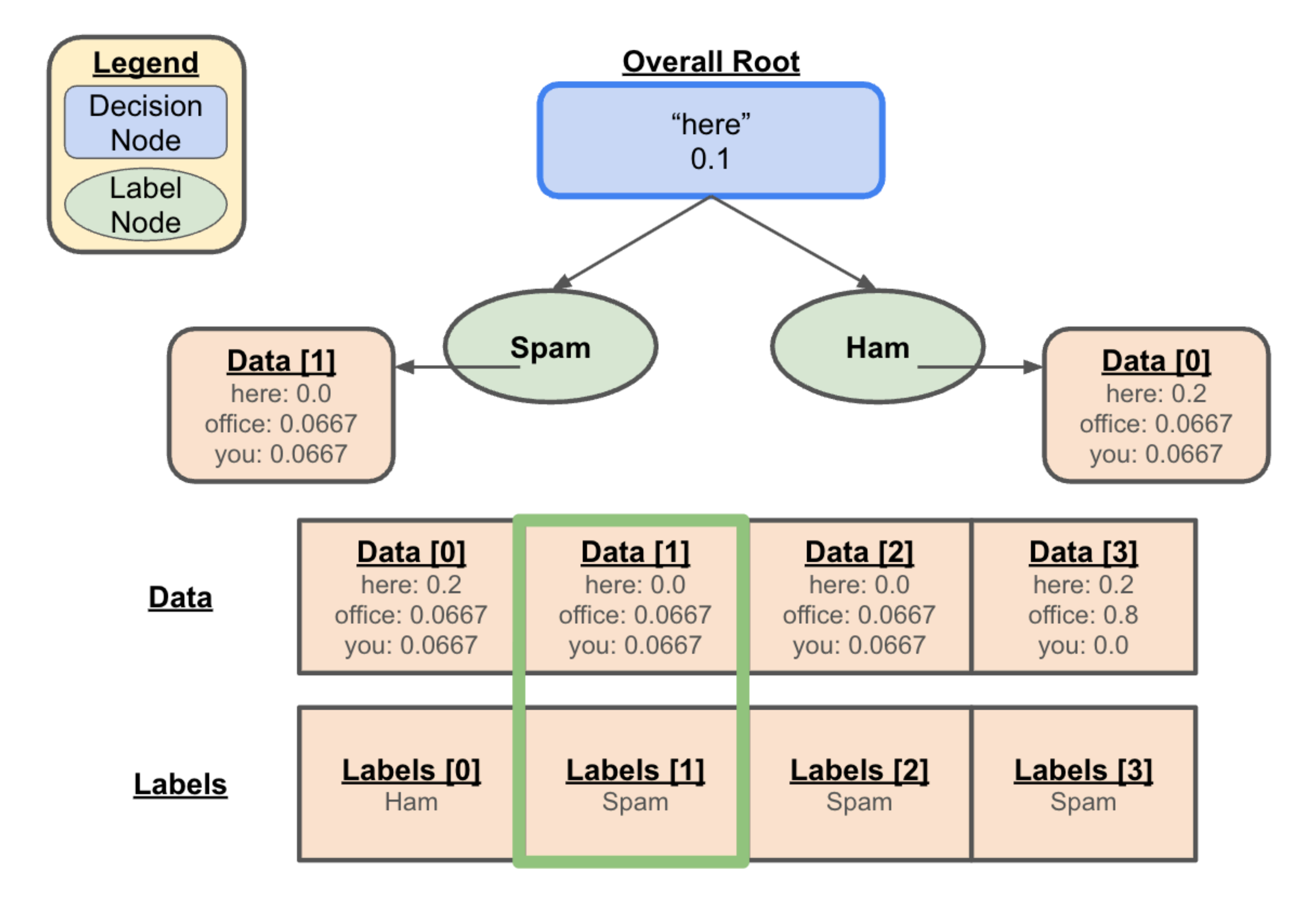
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item:** Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Notice that the label node with `data.get(1)` input is placed to the left of our new decision node because its `TextBlock` has a word frequency of `0` for `"here"` which is less than `0.1` On the other hand, notice that the label node with `data.get(0)` input is placed to the right of the new decision node because its `TextBlock` has a word frequency of `0.2` for `"here"`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- **Current item:** Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- **Current item:** Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

Now we've correctly updated our model to be able to correctly classify the data up to this point!

SIDE NOTE: This algorithm requires you to keep track of both the label and the `TextBlock` datapoint first assigned to this label within every leaf node created in this constructor, as without the previous `TextBlock` datapoint we would be unable to create a new decision node! Ideally we'd like to keep track of all input data that falls under a specific leaf node such that when creating a new decision node, 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.

Step 3: Repeat

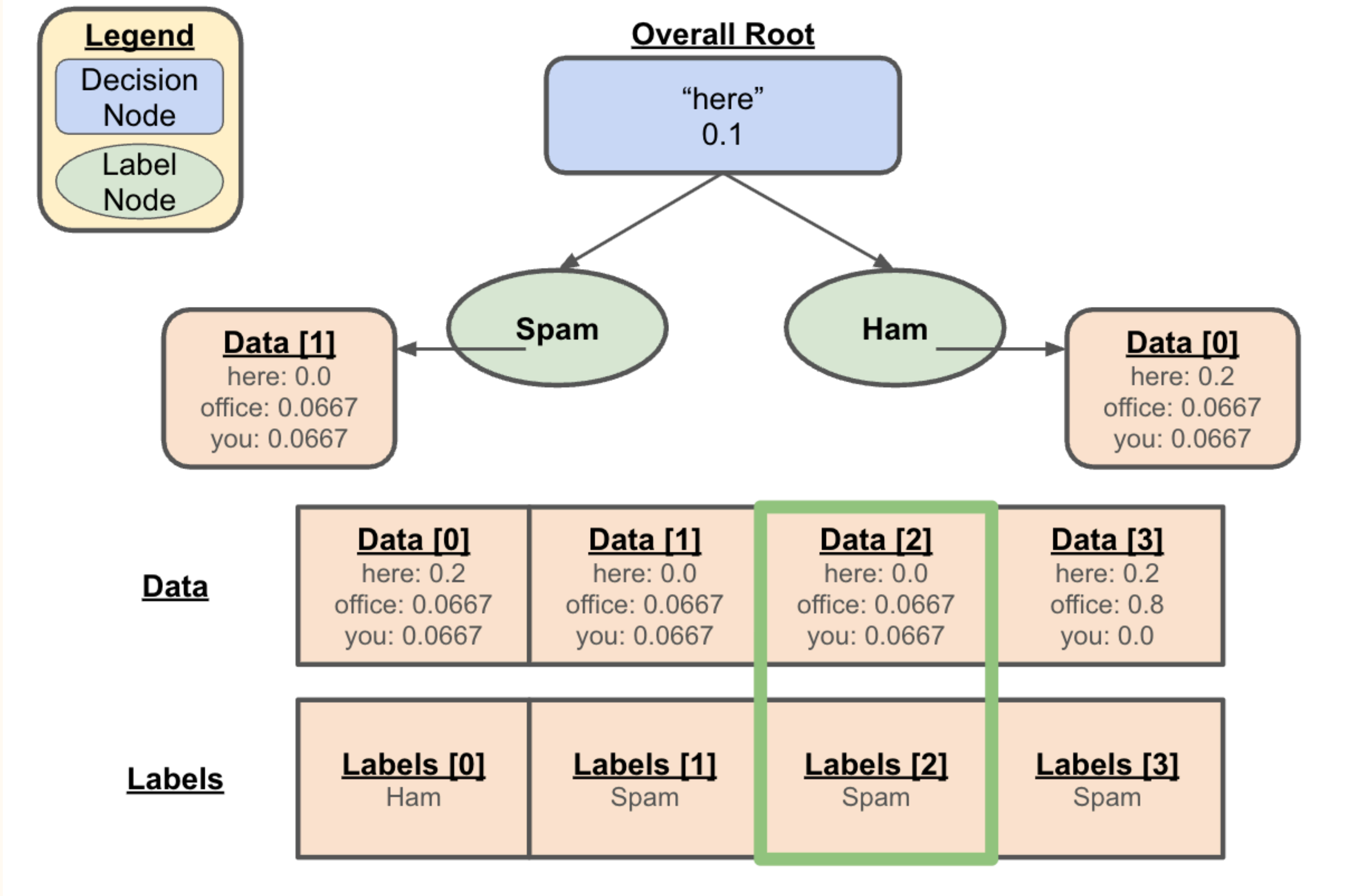
Repeat step 2 for the rest of the list until we've finished processing the list. At that point, our model is fully trained on our data and is able to predict the right label for the data we input.

Expand to see visualization:

▼ Expand

NOTE: The `TextBlock` objects in `data` does store all their respective words from the file. However, we only chose to depict word probabilities that'll be used in the examples for the sake of brevity.

We're now processing index `2`. Start from the root of the tree and traverse through the existing tree until we reach a leaf node.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

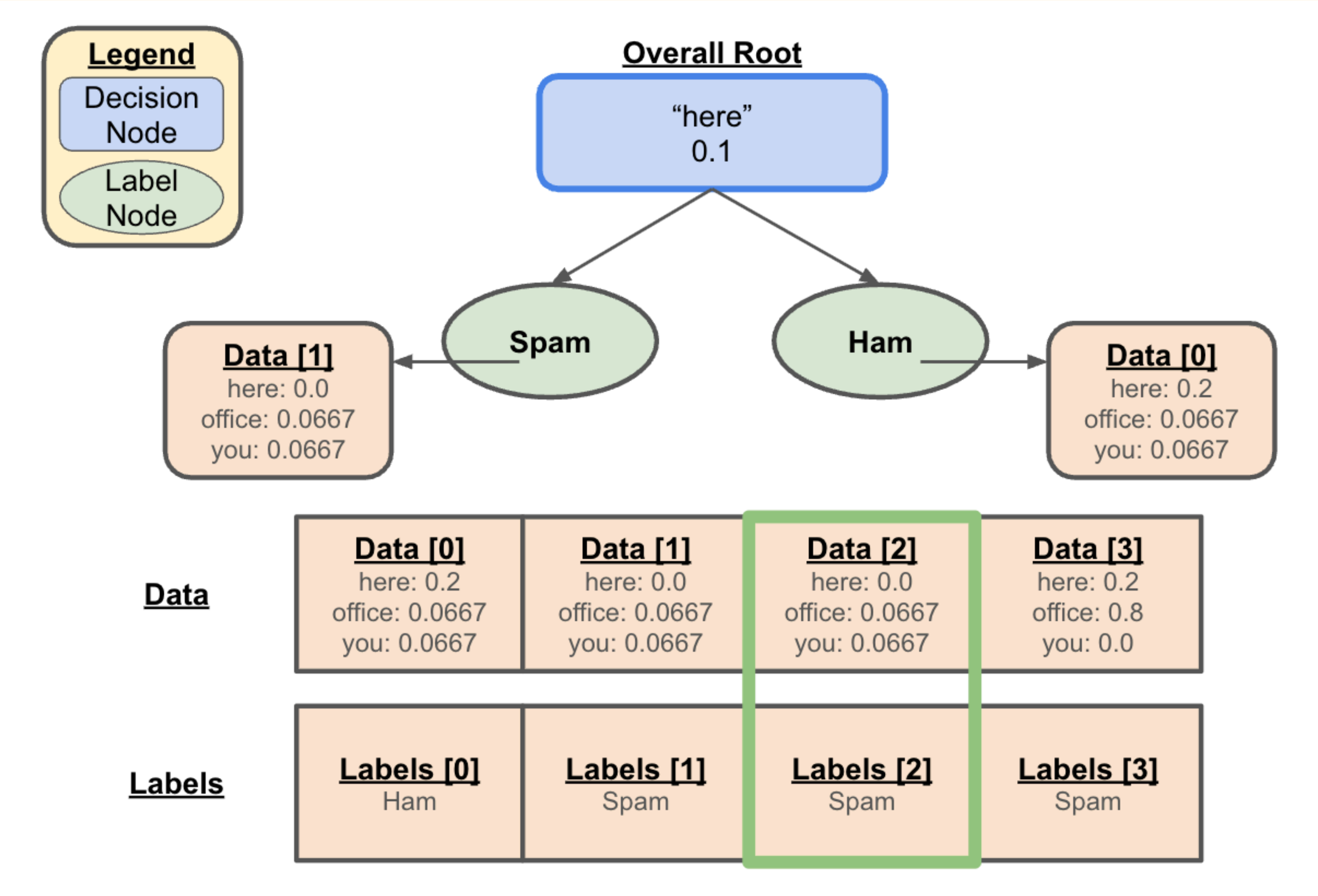
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

Since this datapoint's `"here"` probability is `0.0` which is less than `0.1`, we travel left:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

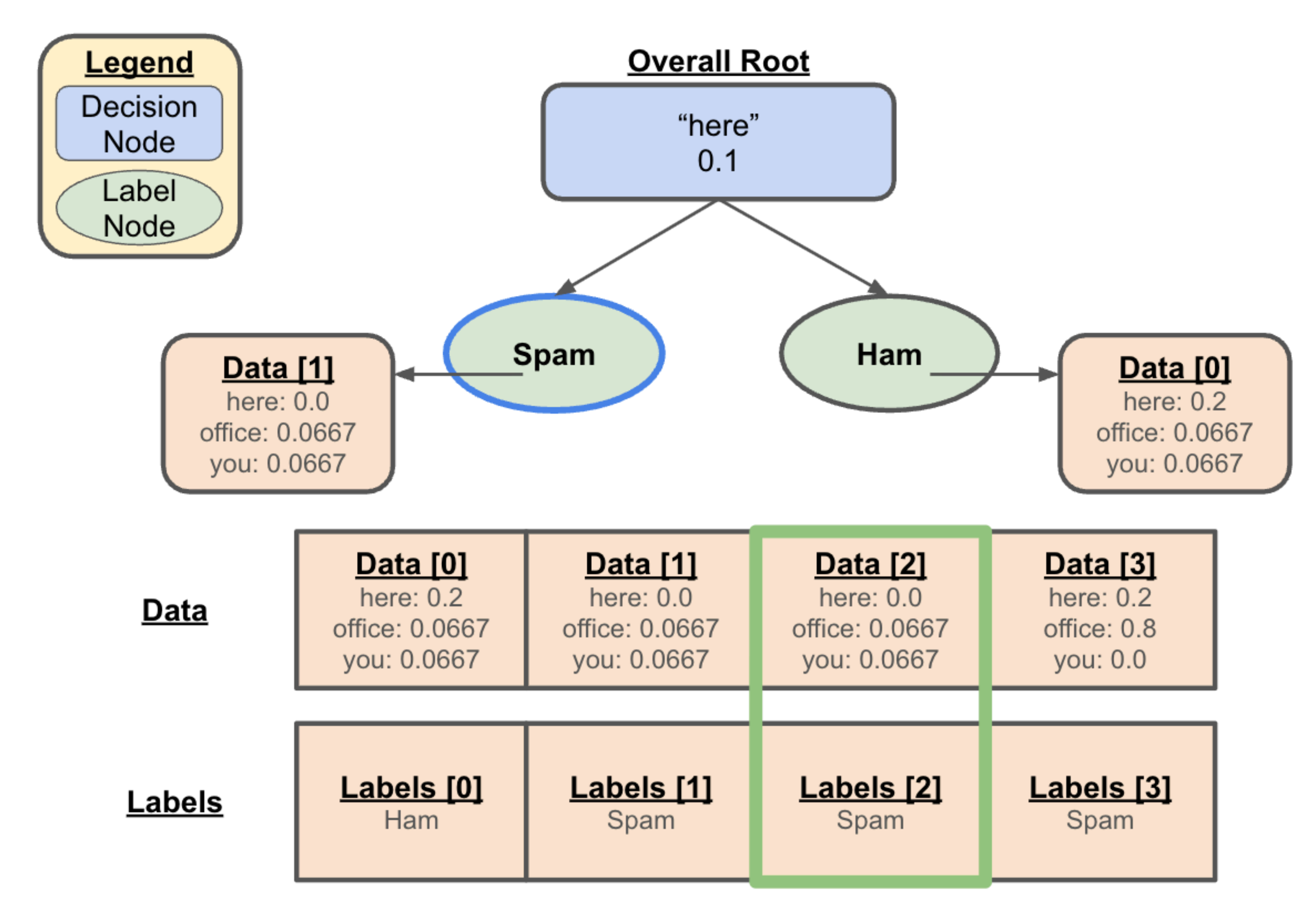
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

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!



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - **current node:** Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

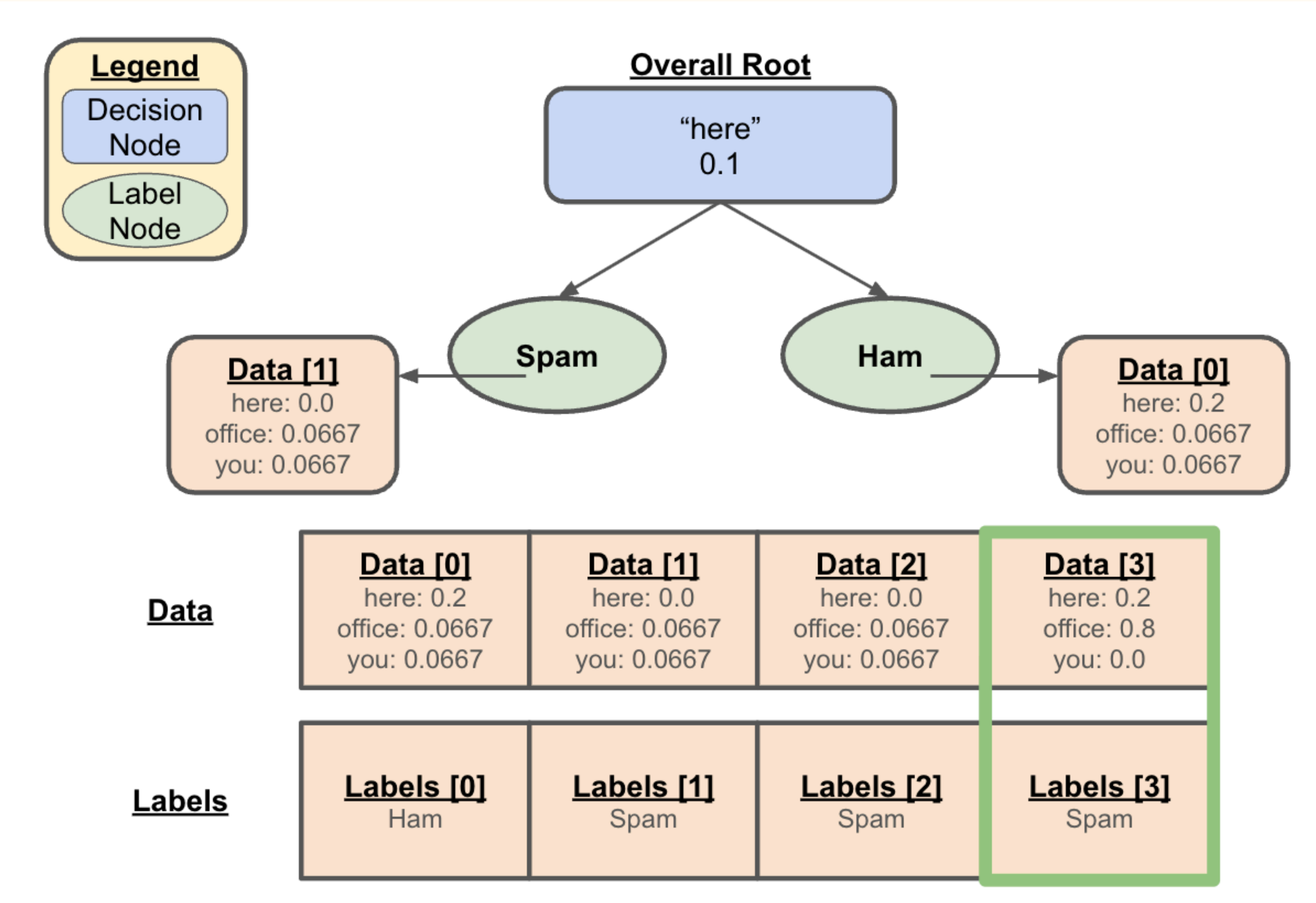
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- **Current item:** Labels[2]: Spam
- Labels[3]: Spam

Lastly, we process index `3`. Start from the root of the tree and traverse through the existing tree until we reach a leaf node.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])

    - Ham (right) (level 1) (stores Data[0])

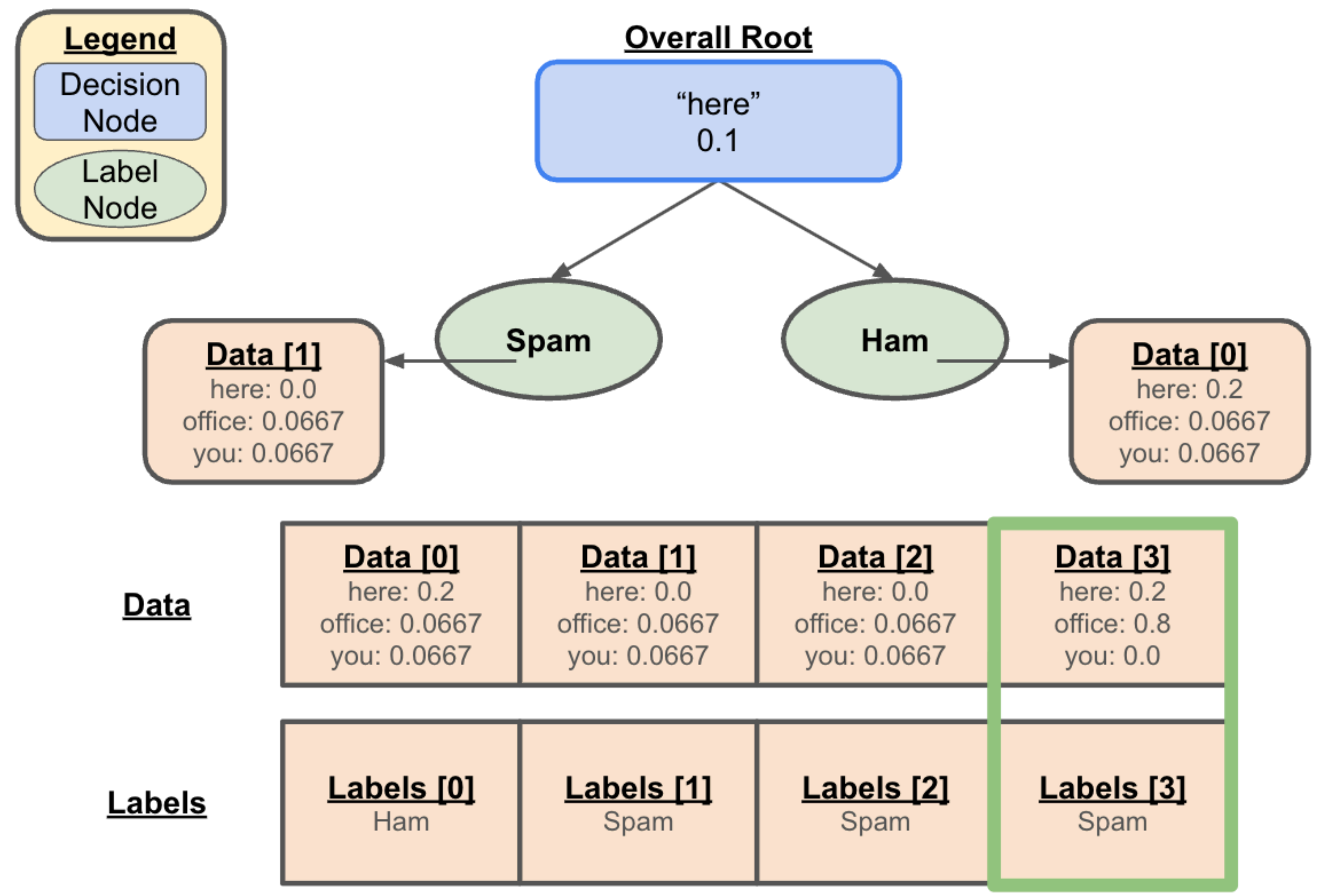
Data List:

- Data[0]:
    - word probability "here" = 0.2
    - word probability "office" = 0.0667
    - word probability "you" = 0.667
- Data[1]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- **Current item:** Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

Since this datapoint's `"here"` probability is `0.2` which is greater than or equal to `0.1`, we travel right:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- **current node:** "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

Data List:

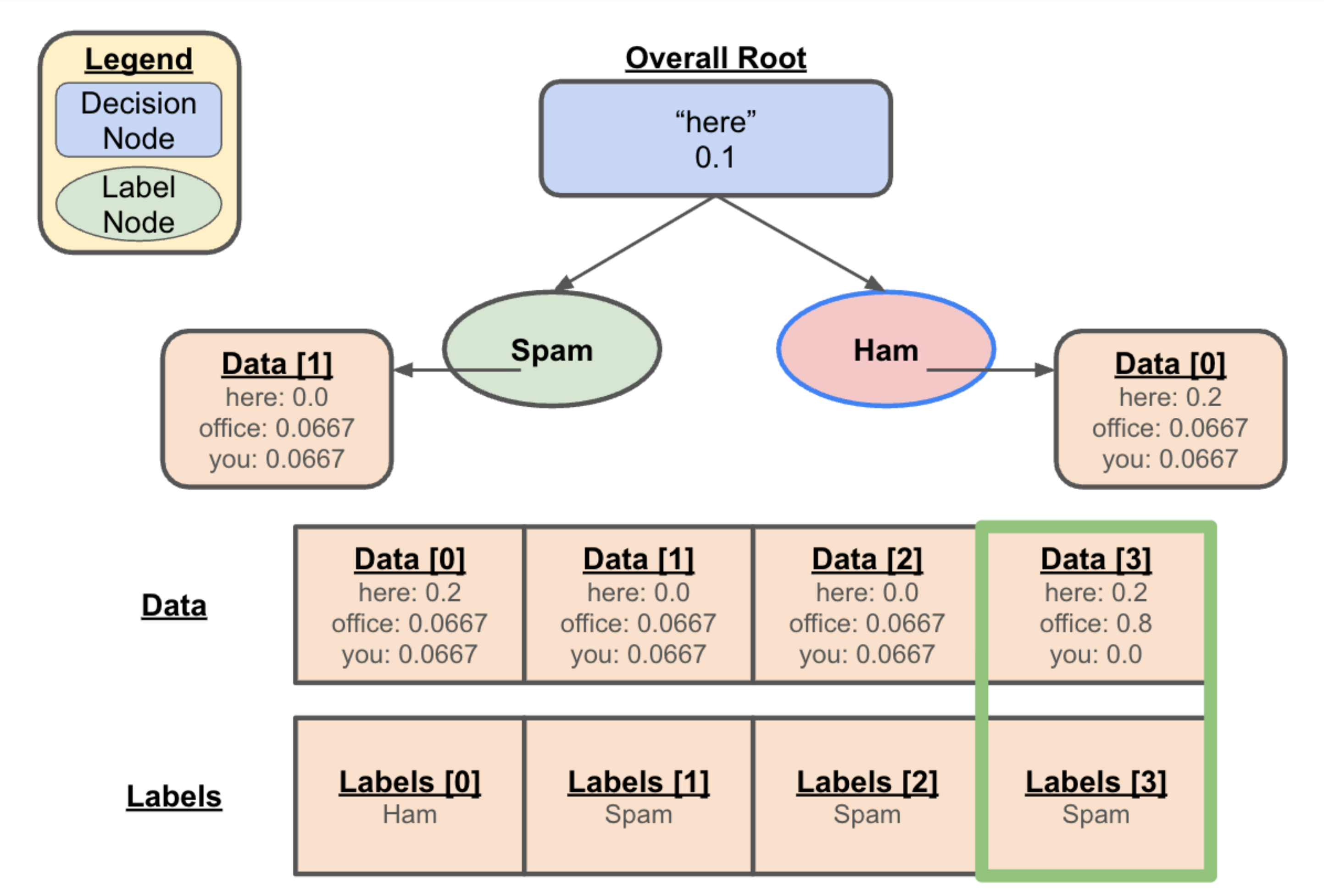
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667

- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

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 decision node with the most differing feature between the two `TextBlock` objects (one previously stored in the `Ham` node, and the other from `Data`) based on their `get()` values:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - **current node:** Ham (right) (level 1) (stores Data[0])

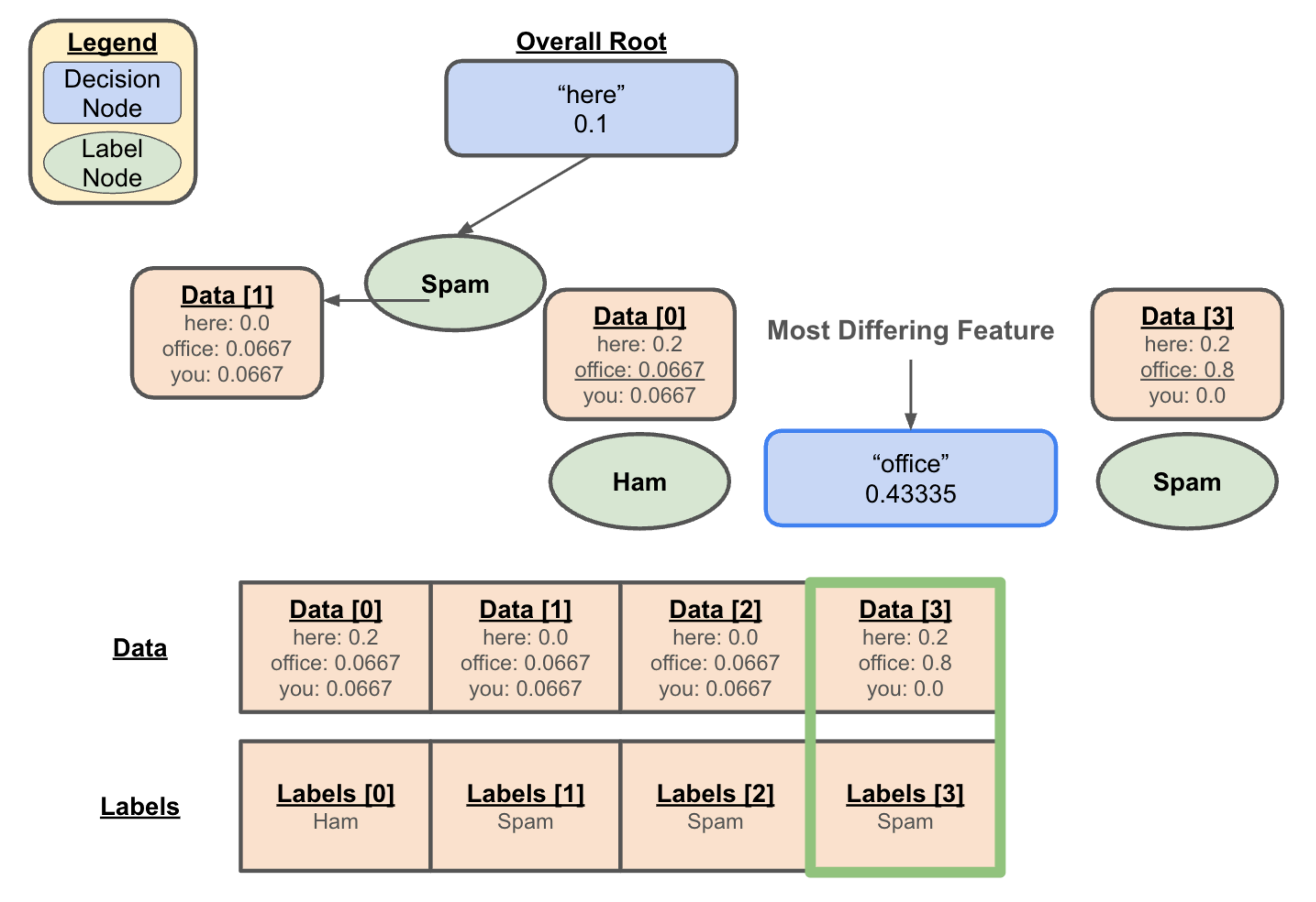
Data List:

- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

We can then utilize the provided methods to produce a new decision node that will allow us to correctly distinguish `data.get(3)` vs. `data.get(0)` using a feature and a threshold based on the algorithm described in **Step 2a**. All that's left to do is organize the label nodes appropriately, as seen below:



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - Ham (right) (level 1) (stores Data[0])

To The Right Of The Tree:

- Ham (stores Data[0])
- **current node:** "office" (middle) threshold = 0.43335
- Spam (stores Data[3])

Data List:

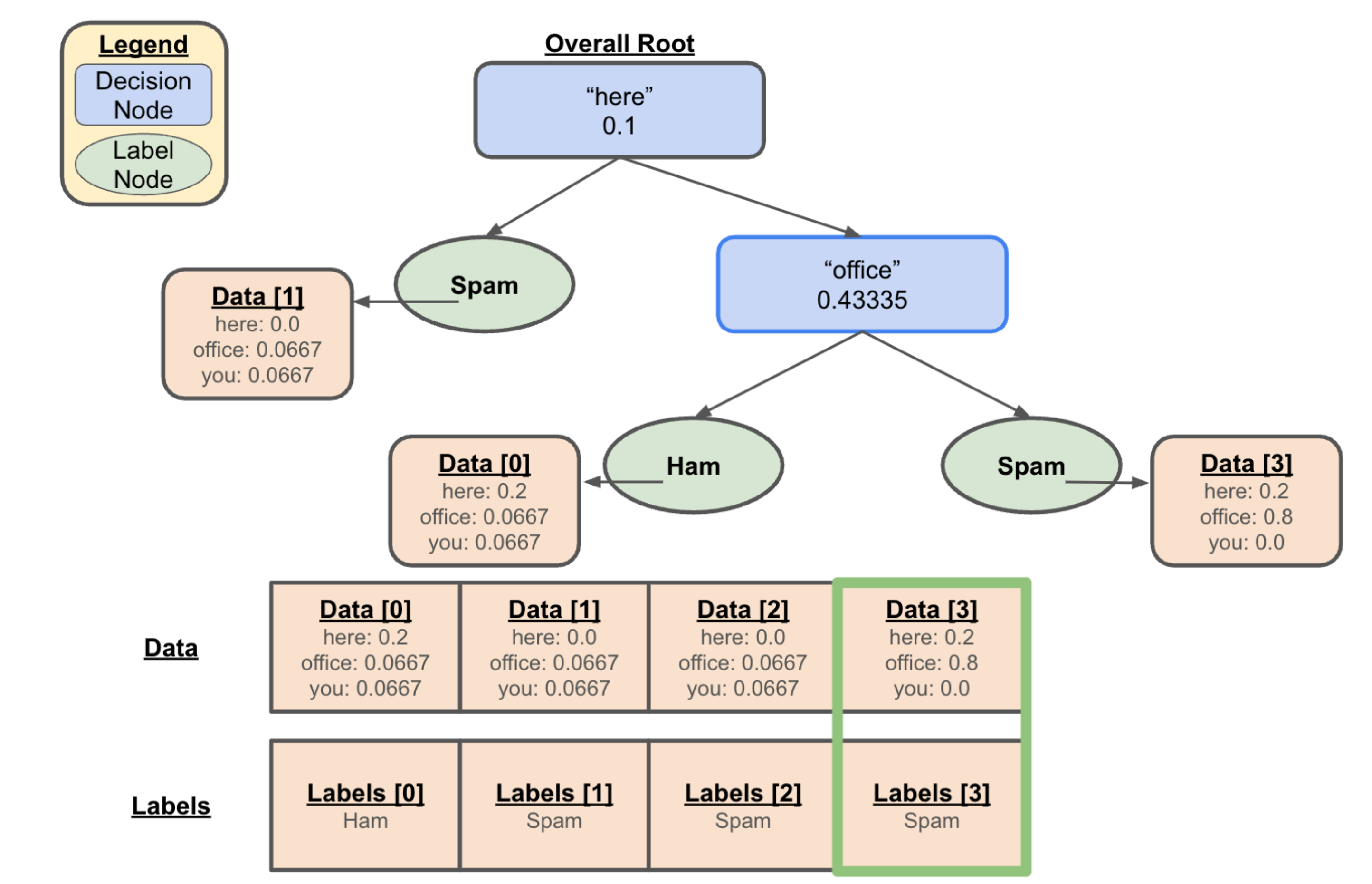
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0

    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- **Current item:** Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

`data.get(0)` is placed to the left of our new decision node because it has a word frequency of `0.0667` for `"office"` which is less than `0.43335,` and `data.get(3)` is placed to the right of the new decision node because it has a word frequency of `0.8` for `"office"`.



Expand to see an alternate equivalent representation of the above image:

▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
  - Spam (left) (level 1) (stores Data[1])
  - **current node:** "office" (right) (level 1) decision node, threshold = 0.43335
    - Ham (left) (level 2) (stores Data[0])
    - Spam (right) (level 2) (stores Data[3])

Data List:

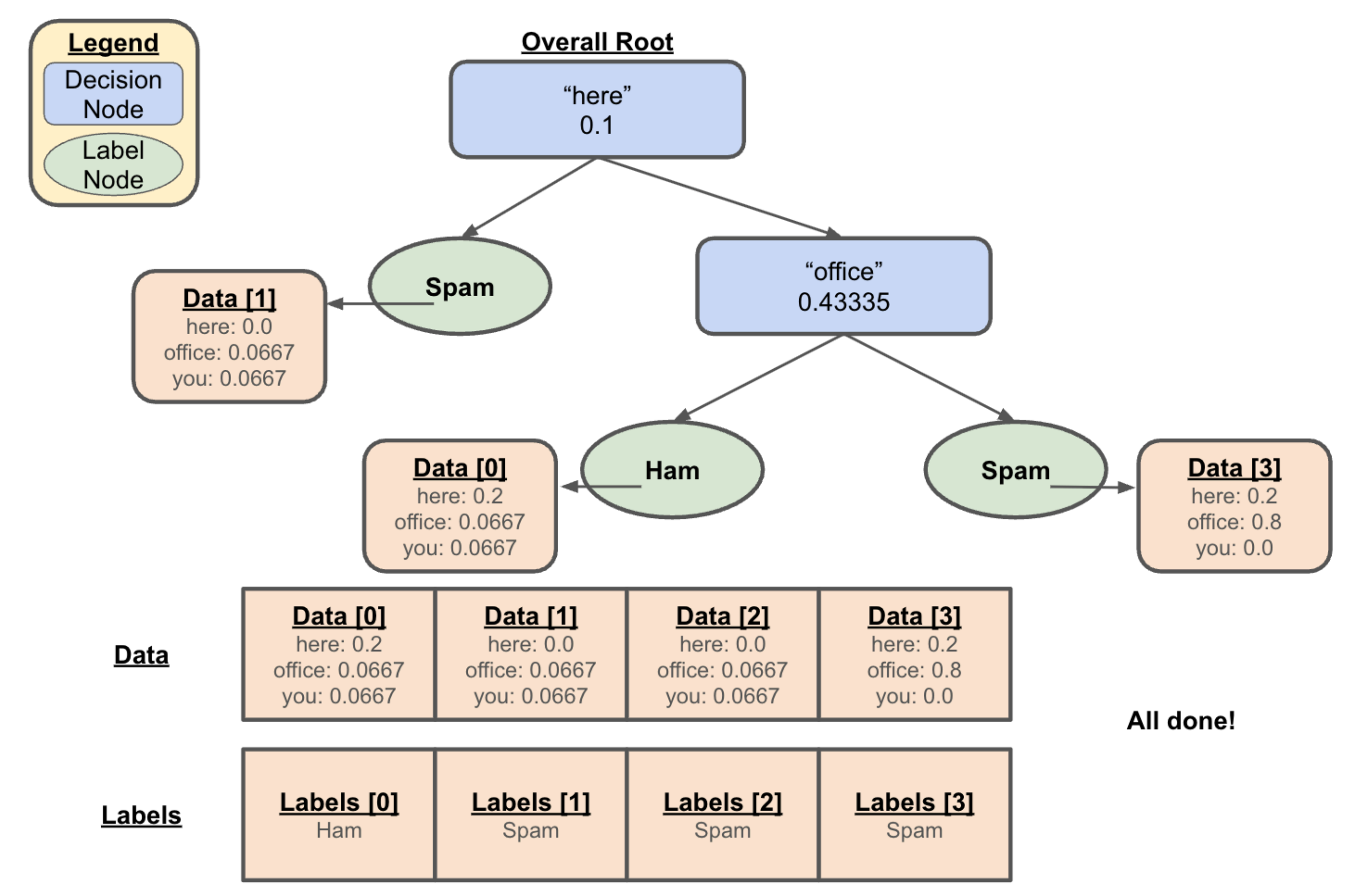
- Data[0]:
  - word probability "here" = 0.2
  - word probability "office" = 0.0667
  - word probability "you" = 0.667
- Data[1]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- Data[2]:
  - word probability "here" = 0.0
  - word probability "office" = 0.0667
  - word probability "you" = 0.0667
- **Current item:** Data[3]:
  - word probability "here" = 0.2
  - word probability "office" = 0.8
  - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- **Current item:** Labels[3]: Spam

We've processed the entire list, so that means we're all done!

Expand to see an alternate equivalent representation of the above image:



▼ Expand

Classification Tree:

- "here" (root) (level 0) decision node, threshold = 0.1
    - Spam (left) (level 1) (stores Data[1])
    - "office" (right) (level 1) decision node, threshold = 0.43335
        - Ham (left) (level 2) (stores Data[0])
        - Spam (right) (level 2) (stores Data[3])

Data List:

- Data[0]:
    - word probability "here" = 0.2
    - word probability "office" = 0.0667
    - word probability "you" = 0.667
- Data[1]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667
- Data[2]:
    - word probability "here" = 0.0
    - word probability "office" = 0.0667
    - word probability "you" = 0.0667

- Data[3]:
    - word probability "here" = 0.2
    - word probability "office" = 0.8
    - word probability "you" = 0.0

Labels List:

- Labels[0]: Ham
- Labels[1]: Spam
- Labels[2]: Spam
- Labels[3]: Spam

To The Right:

All done!

HINT from Cornbear: This algorithm requires you to keep track of the initial `TextBlock` used to create the label node. Without this initial `TextBlock`, we would be unable create a new feature for when our model is inaccurate! Keeping this in mind, what may be one of the fields needed in the `ClassifierNode` class?

HINT from Cornbear: It looks like we're processing data and using that information to *modify* our tree. Keeping in mind our recently learned concept, **what pattern should we employ to help implement this constructor?**

At this point, test your current implementation. Once these tests are passing, the assignment should be completed. CONGRATULATIONS!!! Cornbear will be thrilled to see what you've made!! Make sure your code adheres to the Code Quality Guide and Commenting Guide that we cover below!

Below, we've provided sample client input and output that should be your expected output at this point (user input is **bold and underlined**).

NOTE: `TRAIN_FILE` and `TEST_FILE` were set to `"data/federalist_papers/train.csv"` and `"data/federalist_papers/test.csv"` respectively:

▼ Expand

```
 _______  ___      _______  _______  _______  ___   _______  ___   _______  ______
|       ||   |    |   _   ||       ||       ||   | |       ||   | |       ||    _ |
|       ||   |    |  |_|  ||  _____||  _____||   | |    ___||   | |    ___||   | ||
|       ||   |    |       || |_____ | |_____ |   | |   |___ |   | |   |___ |   |_||_
|      _||   |___ |       ||_____  ||_____  ||   | |    ___||   | |    ___||    __  |
|     |_ |       ||   _   | _____| | _____| ||   | |   |    |   | |   |___ |   |  | |
|_______||_______||__| |__||_______||_______||___| |___|    |___| |_______||___|  |_|
Welcome to the CSE 123 Classifier!
(Remember to edit the TRAIN_FILE and TEST_FILE class constants if you want to change the files b

To begin, enter your desired mode of operation:
```

```
1) Train classification model (Two List Constructor)
2) Load model from file (Scanner Constructor)
Enter your choice here: 1

Would you like to shuffle the data?
1) Yes (Recommended for testing finalized models)
2) No  (Recommended for debugging models)
2

What would you like to do with your model?

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 2
Overall: 0.6923076923076923
MADISON: 0.6666666666666666
JAY: 1.0
HAMILTON: 0.6666666666666666

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 3
Would you like to save to a file or output the classification tree to console?
1) Save to a file
2) Output classification tree to console
2

Save output:

Feature: and
Threshold: 0.038483040043334985
Feature: be
Threshold: 0.0233106848494183
Feature: in
Threshold: 0.025932789896541807
Feature: of
Threshold: 0.0657510278337315
Feature: of
Threshold: 0.059036768121312144
Feature: by
Threshold: 0.012555871395333838
Feature: the
Threshold: 0.09027856650690169
Feature: be
Threshold: 0.013843510911220169
"MADISON, HAMILTON"
HAMILTON
Feature: to
Threshold: 0.03464681930254901
```

```
MADISON
HAMILTON
MADISON
Feature: the
Threshold: 0.08925919990423102
Feature: the
Threshold: 0.0797962395506934
HAMILTON
MADISON
Feature: to
Threshold: 0.04275799174944023
Feature: be
Threshold: 0.010925081284708565
"MADISON, HAMILTON"
MADISON
HAMILTON
Feature: executive
Threshold: 0.009212885714106266
Feature: of
Threshold: 0.07337675492176134
MADISON
HAMILTON
MADISON
HAMILTON
Feature: the
Threshold: 0.08428712253209539
Feature: in
Threshold: 0.017843227023234286
MADISON
Feature: would
Threshold: 0.0066798009567743035
MADISON
HAMILTON
Feature: of
Threshold: 0.061990537892121986
Feature: to
Threshold: 0.042701591658820356
MADISON
HAMILTON
HAMILTON
Feature: the
Threshold: 0.07024915442197364
JAY
Feature: be
Threshold: 0.012752794487431048
"MADISON, HAMILTON"
MADISON

1) Test with an input file (classify)
2) Get testing accuracy (calculateAccuracy)
3) Save classification tree (save)
4) Quit
Enter your choice here: 4
```

Relevant Problems:

- Section 14: Remove Leaves in List
- Section 14: Make Full

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 creating decisions 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).

Code Quality

> ⚠ **NOTE:** To earn a grade higher than N on the Behavior and Concepts dimensions of this assignment, **your *core* algorithms for each method in `Classifier` must be implemented *recursively*. You will want to utilize the *public-private pair* technique discussed in class.** You are free to create any helper methods you like, but the core of your implementations must be recursive.

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 manually set fields of an object immediately after construction (when possible) — there should be a constructor included for this situation.
    - For example, if you were the implementor of the `Point` class:

```
Point coord = new Point(); // Poor usage of constructor
coord.x = 5; // ✕
coord.y = 7; // ✕

Point coord = new Point(5, 7); // ▫ Correct usage of constructor
```

- **Methods:**
  - All methods present in `Classifier` that are not listed in the specification must be `private`.
  - Make sure that all parameters within a method are used and necessary.
  - When designing helper methods, avoid unnecessary returns.

- **`x = change(x)`:**
  - Similar to linked lists, do not "morph" a node by directly modifying fields (especially when replacing a branch 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 when 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 `Classifier` class private. (Fields in your `ClassifierNode` 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.