

Programming Assignment 3: Spam Classifier

Background

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

Terminology

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

- **Model:** The actual program that makes probabilistic classifications on provided inputs.
- **Training:** Models are "trained" on previously gathered datasets to make future predictions.
- **Label:** How data is classified after being run through the model. In our tree, leaf nodes will house classification labels.
- **Feature:** Important aspects/characteristics of our dataset that we use in classification that correspond to a numeric value. Typically, the hardest part of a machine learning algorithm is determining how to take input data and "featurize" it into something a computer can understand
 - Ex: turning a sentence or image into a series of numbers.
- **Threshold:** The numeric value we 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.

Specification

Learning Objectives

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

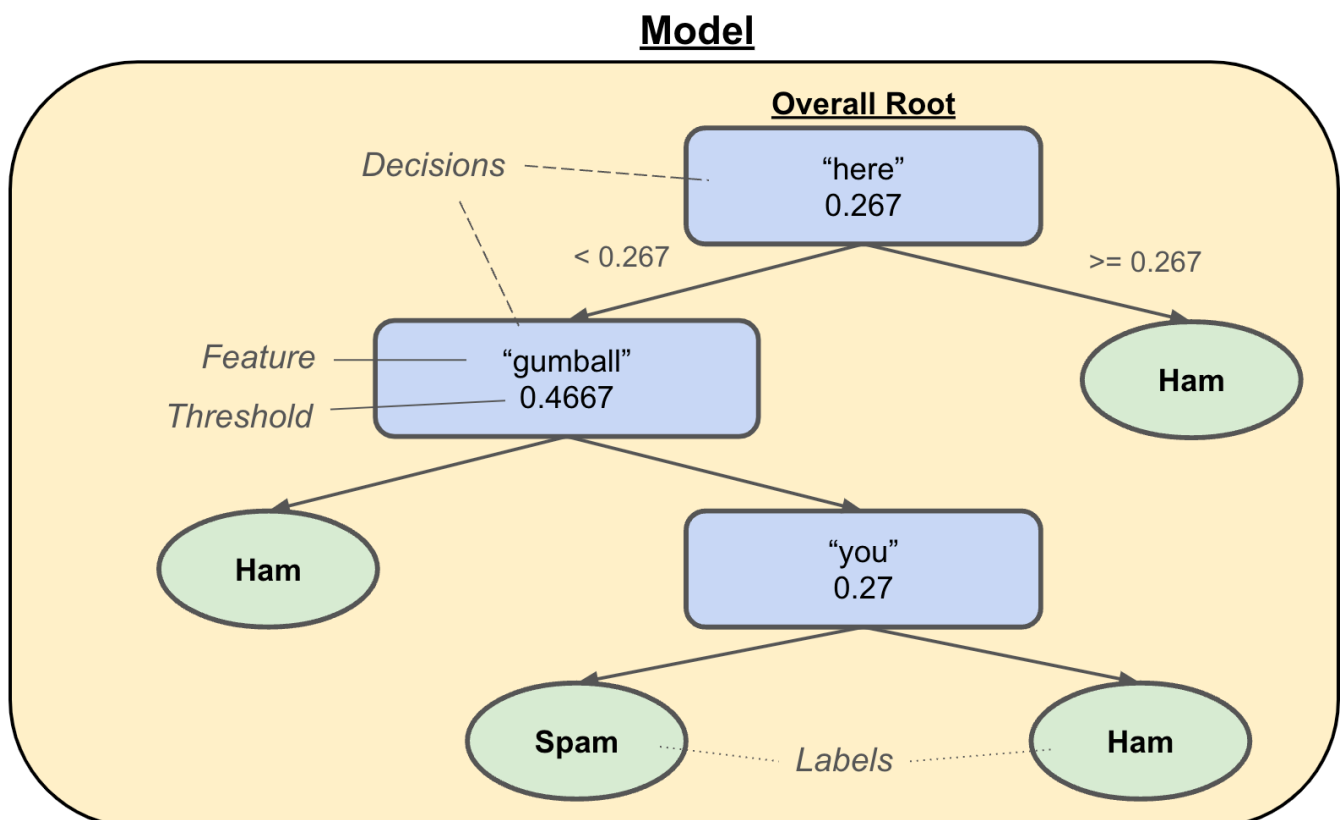
- Define data structures to represent compound and complex data
- Write a functionally correct Java class to represent a binary tree.
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, implementation, and performance.
- Produce clear and effective documentation to improve comprehension and maintainability of programs, methods, and classes.

Assignment



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

Your goal for this assignment is to implement a classification tree, a simplistic machine-learning model that predicts a label when given some text-based data. Below is a visual example of what a classification tree might look like for classifying spam emails. It also includes relevant labels for each of the vocab terms defined on the last slide.



As seen above, in our classification tree the **leaf nodes represent our predictive labels** ("Spam" or "Ham" – a funny way of writing not spam) while the **intermediary nodes represent decision nodes** with some feature of our data (the word probability of a word in the email) using a threshold to determine what decision to make. To reach a classification for some input, you start at the root of the tree and determine whether the corresponding feature falls to the left or right of the current node's threshold (determined by `<` or `>=`) and travel in the corresponding direction. Repeating this process will eventually lead you to a classification for your input.

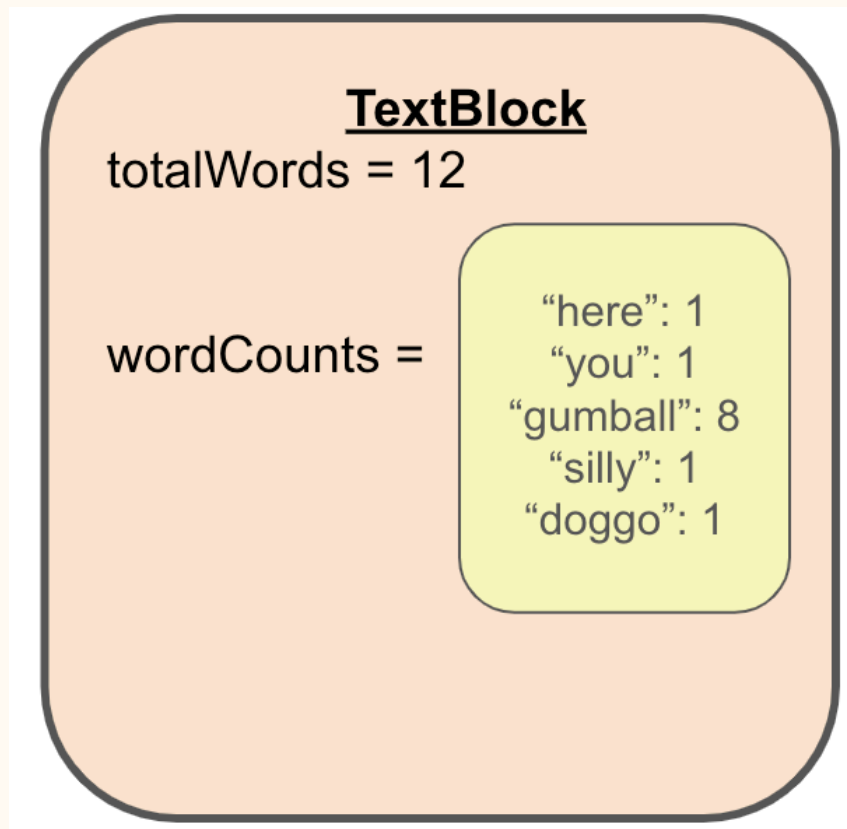
Below we'll trace through the classification of a sample input with our example model shown above.

▼ Expand

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

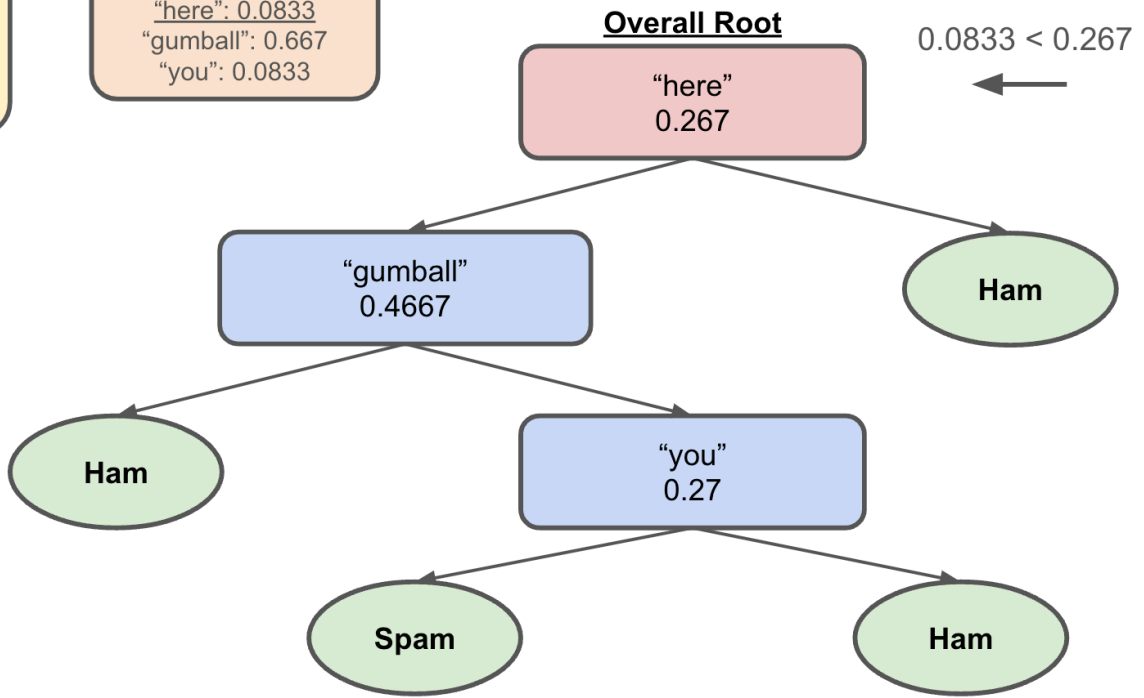
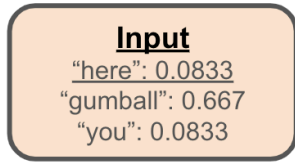
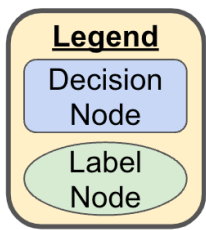
```
here gumball gumball gumball gumball you silly gumball gumball gumball gumball doggo
```

producing the following `TextBlock` object:

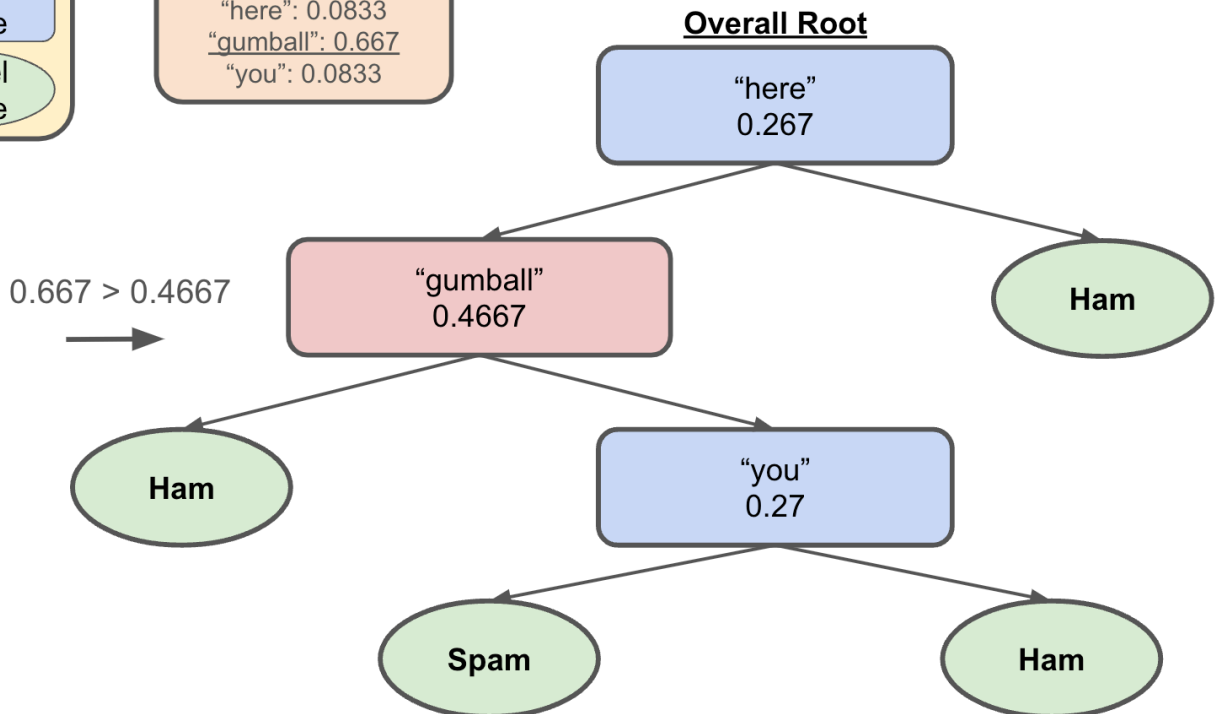
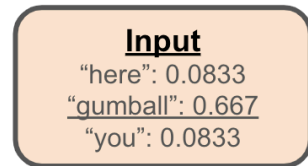
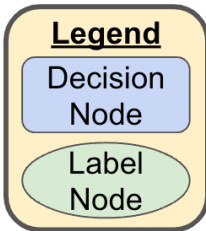


Above, we see that our `TextBlock` object has `totalWords=12`, as well as a mapping `wordCounts` of a word to its count. In this example, "here", "you", "silly", "doggo" occurs once in the content, hence the mapping of those words to the occurrence of "1". However "gumball" appears eight times in the provided content, hence the mapping of "gumball" to "8".

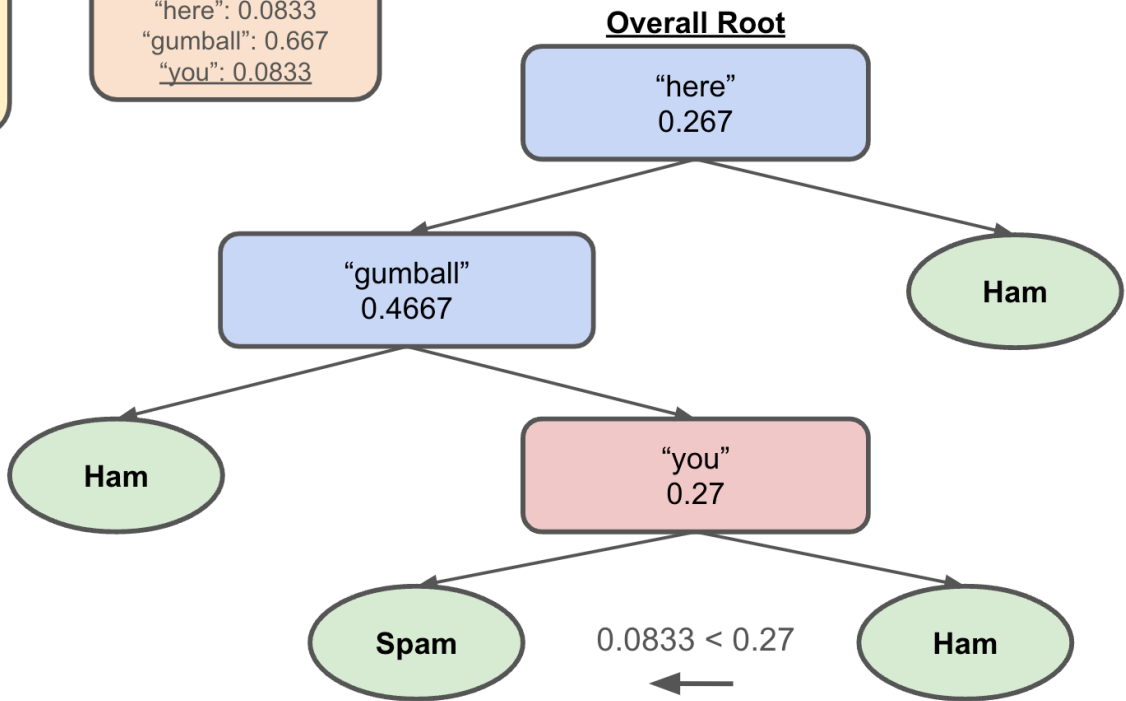
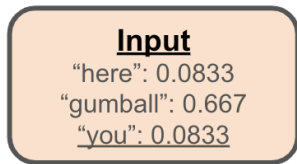
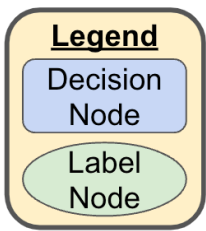
1. Since the word probability of "here" in our `TextBlock` (0.0833) is less than the threshold (0.267), we travel left of the "here" node



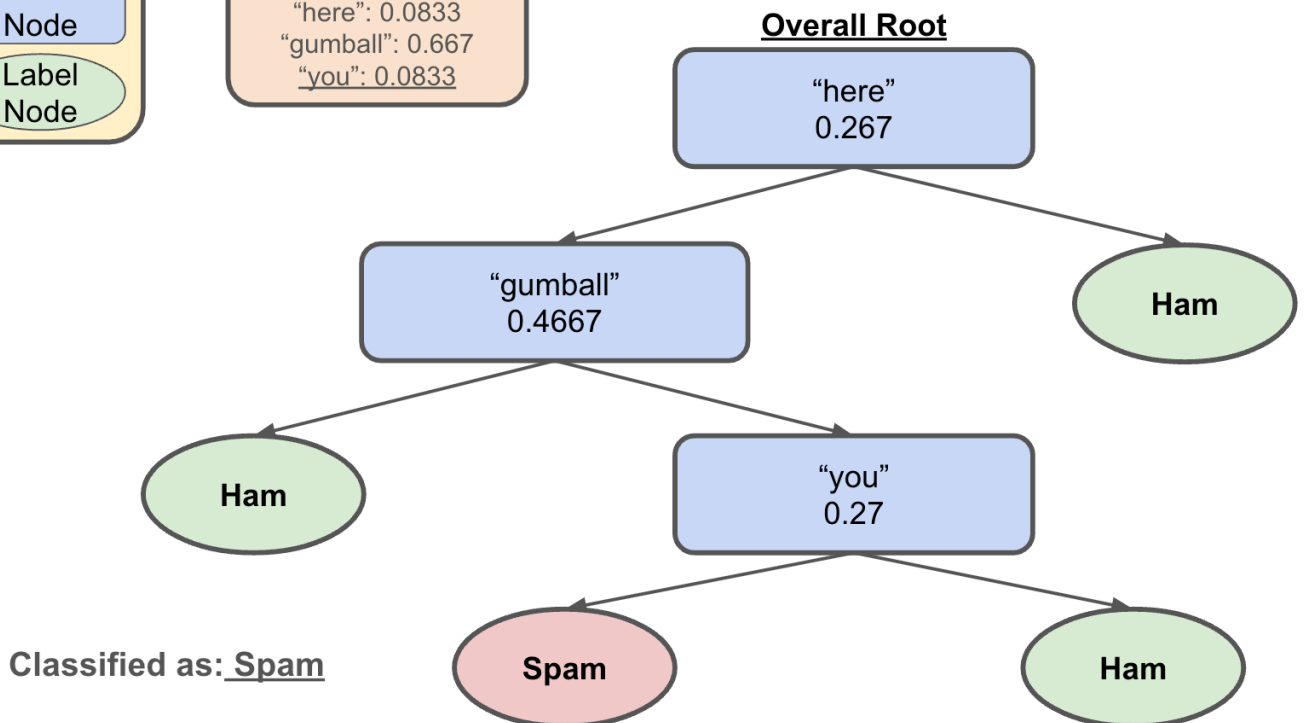
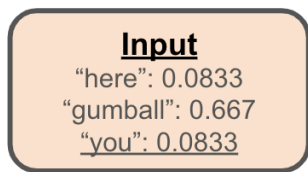
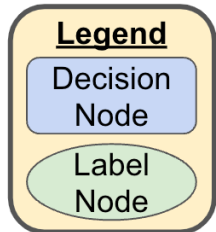
2. Since the value of the "gumball" feature in our `TextBlock` (0.667) is greater than or equal to the threshold (0.4667) we'll travel right to the "you" node



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

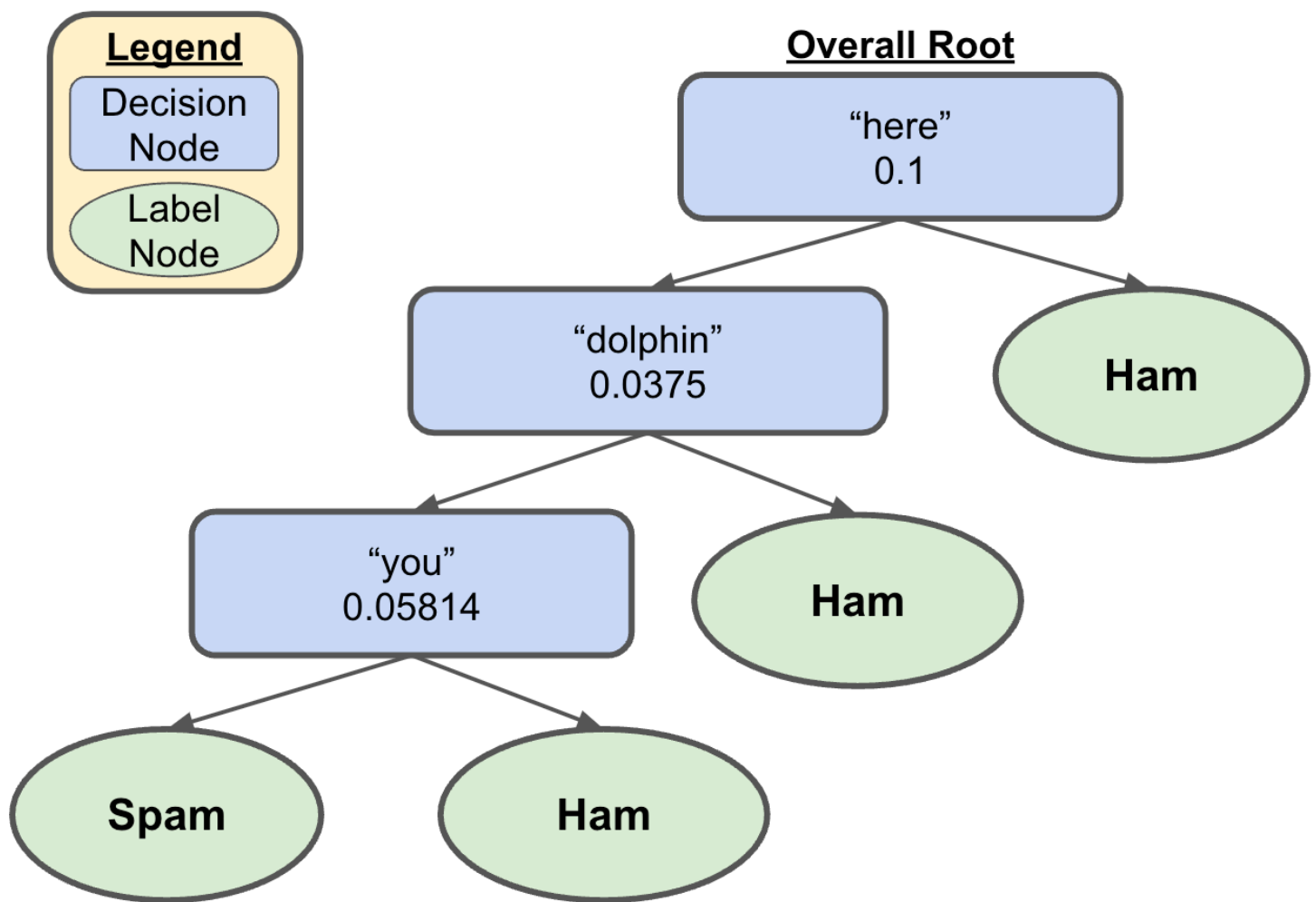


4. We have reached a leaf node and therefore can predict that input corresponds to "Spam" — a spam email (the resulting label)



These classification trees may not always be the same. Below is an alternative example of what a

potential classification tree might look like for emails.



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:

content

"hello, i am here at your office but the door is locked. are you there?"

TextBlock

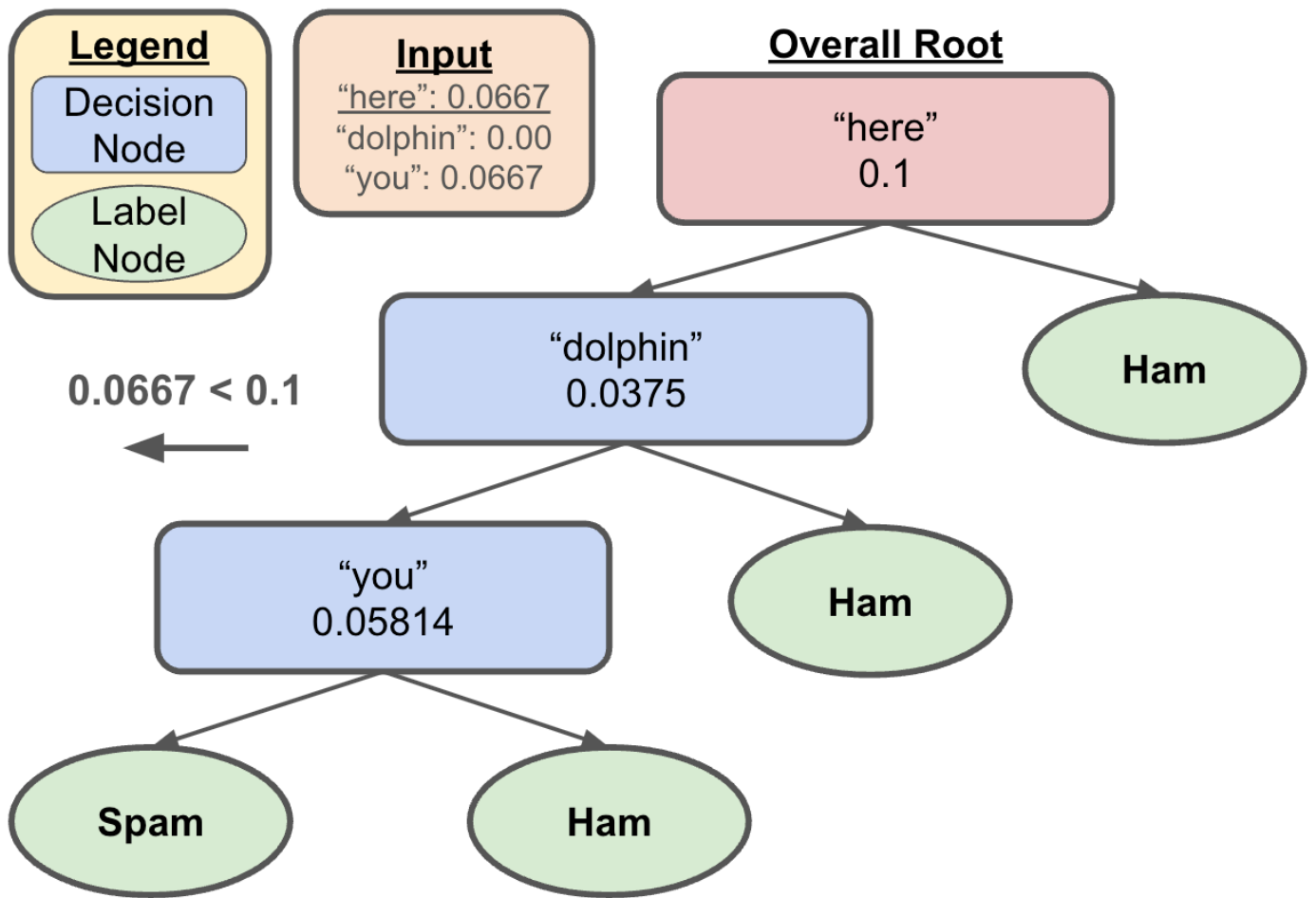
totalWords = 15

wordCounts =

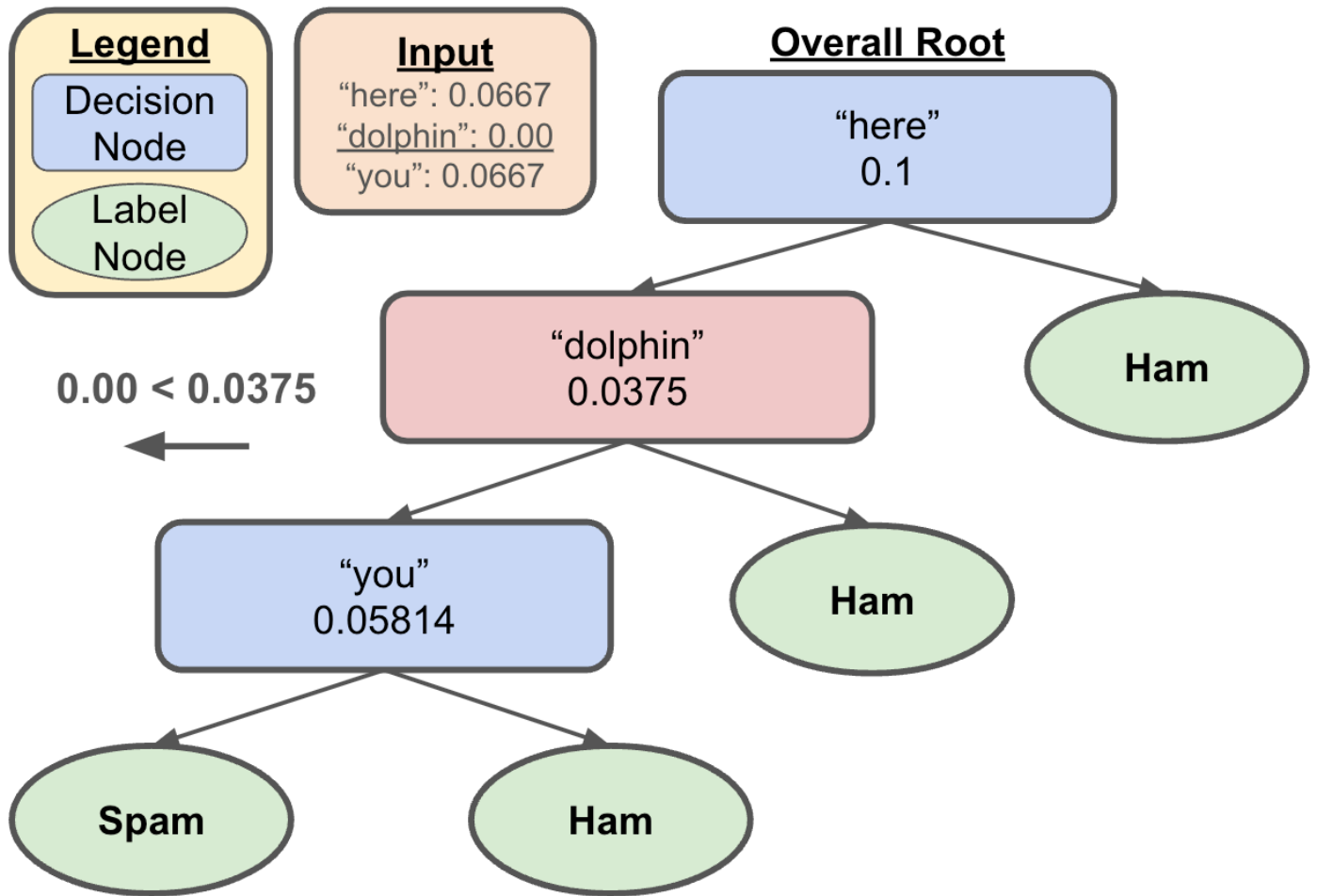
"here": 1
"you": 1
"hello": 1
"i": 1
"am": 1
"at": 1
"your": 1
"office": 1
"but": 1
"the": 1
"door": 1
"is": 1
"locked.": 1
"are": 1
"there": 1

In this example, each of the words in the input "hello, i am here at your office but the door is locked. are you there?" occur once. Therefore, the total number of words is 15, and the `wordCounts` mapping maps each word to an occurrence of 1.

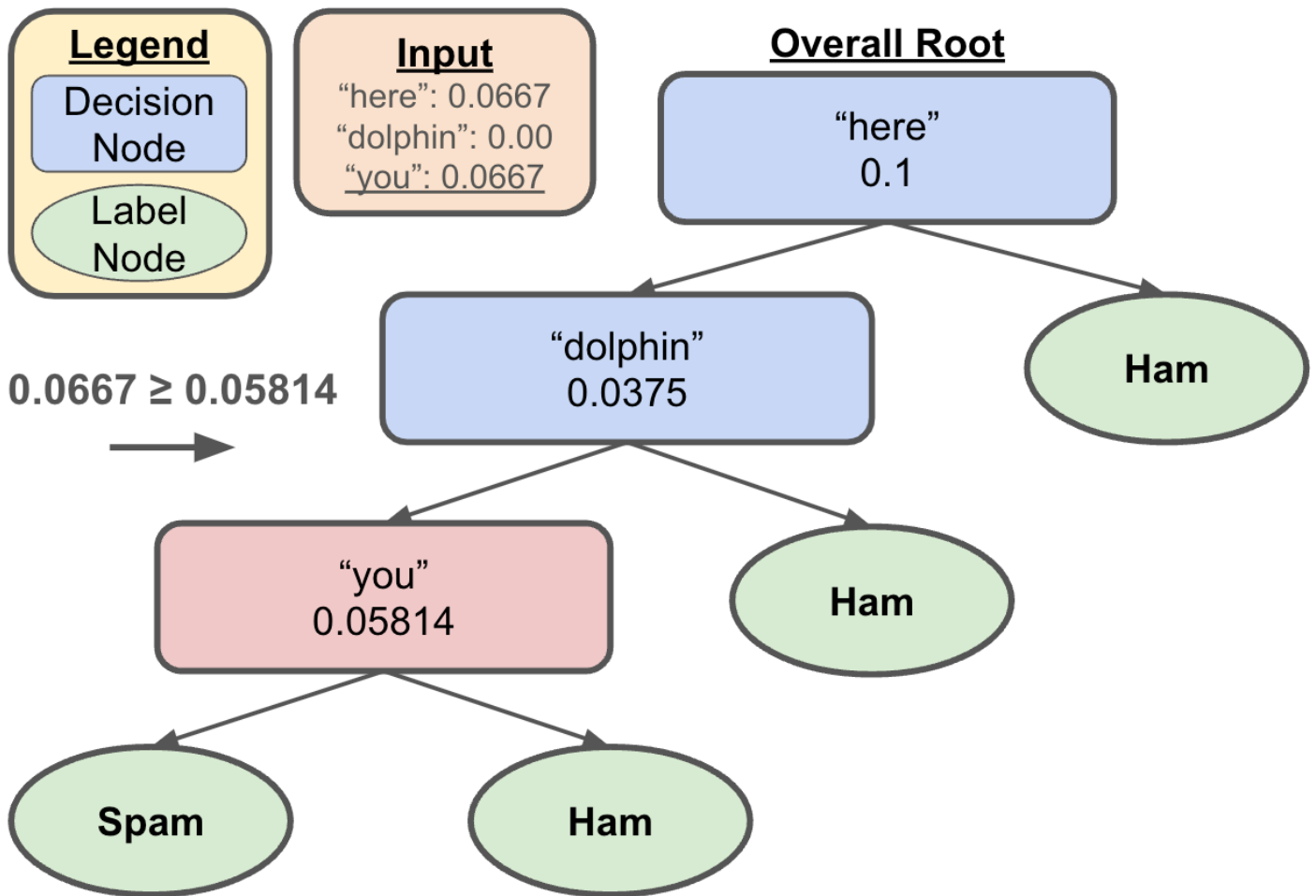
1. Since the word probability of "here" is 0.0667, which is less than the threshold (0.1) we'll travel left to the "dolphin" node.



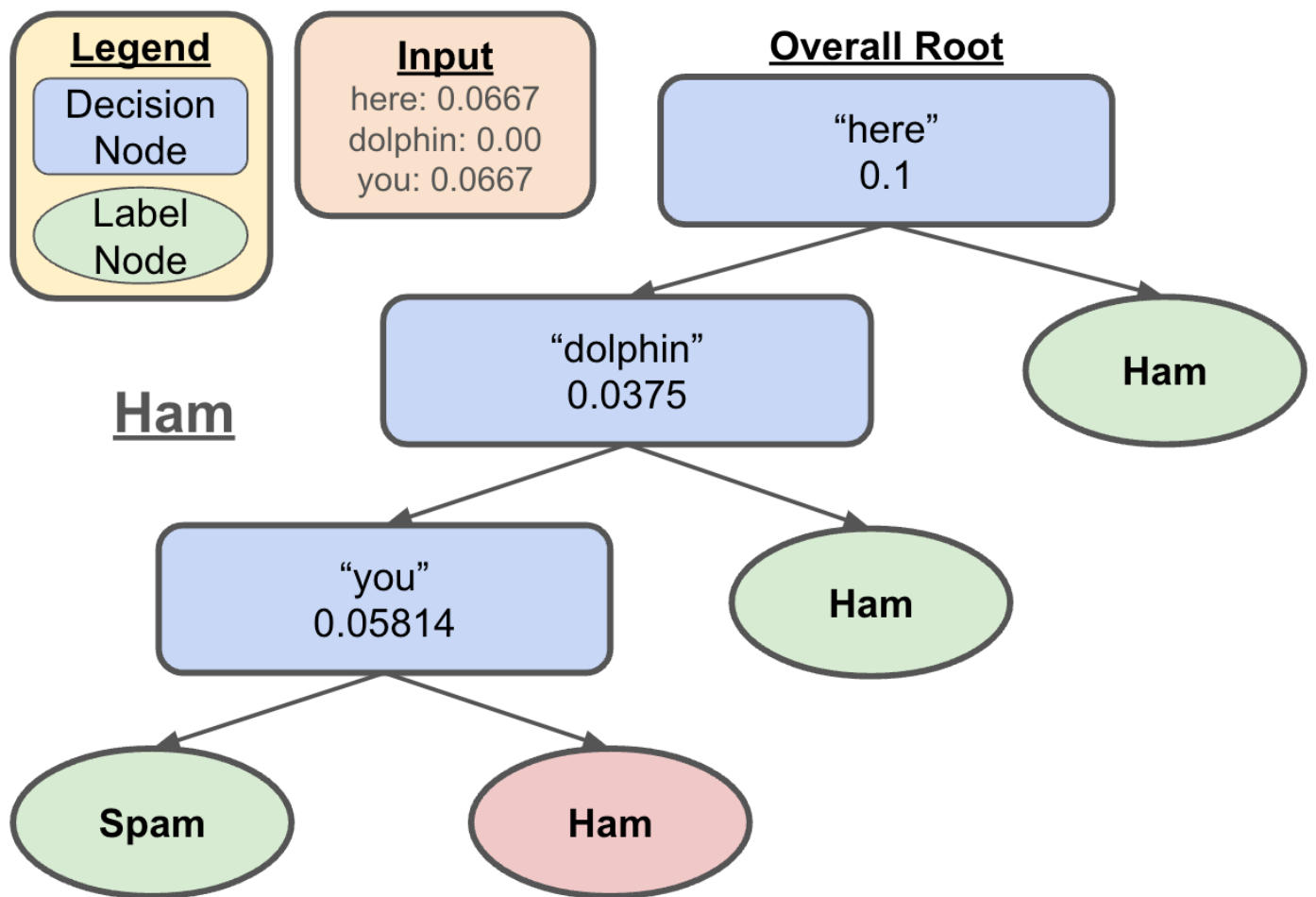
2. Since the word probability of "dolphin" is 0.00, which is less than the threshold (0.0375) we'll travel left to the "you" node.



3. Since the word probability of "you" is 0.0667, which is greater than or equal the threshold (0.05814) we'll travel right to the Ham node.



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



This is what you'll be implementing in this assignment! Specifically, you'll be creating a classification tree that's able to predict some 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 for a [Federalist Paper](#)!

System Structure

Below we describe the **provided** `TextBlock` class that will be used in your implementation of `Classifier.java`. 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 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 Operations

Classifier.java

For this assignment, you're **only** required to implement `Classifier.java`. Below are the methods you should implement:



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.

```
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.
 - 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.

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

- Create a classifier from the input data and corresponding labels.
 - **Note that you are building the tree up from scratch in this constructor.**
- The lists should be processed in parallel 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 below:
 - For a specific index, traverse through the current classification tree (the tree you are currently building) until you reach a leaf node.

- If the node's label matches the current label, do nothing (our model is accurate up to this point).
- If the label is incorrect, create a new decision node with the `TextBlock` data used to create the original leaf node* and our current `TextBlock` input. The decision node should be placed where the original leaf node currently is.
 - You should use the `findBiggestDifference` method within the `TextBlock` class to find the feature and generate the new midpoint as the threshold.
 - After the new decision node is constructed, the original leaf node and current input should be placed appropriately.
- 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

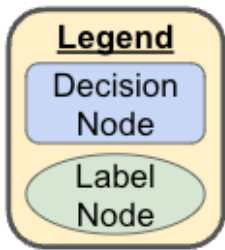


* This algorithm requires you to keep track of the initial `TextBlock` used to create the label node. Without this initial `TextBlock`, we would be unable to create a new feature for when our model is inaccurate! Keeping this in mind, what may be one of the fields needed in the `ClassifierNode` class?

The algorithm above is further shown in the following diagrams with `Data` being the `List<TextBlock>` and `Labels` being the `List<String>`:

▼ Expand

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

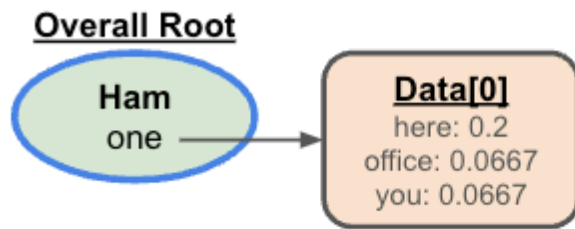
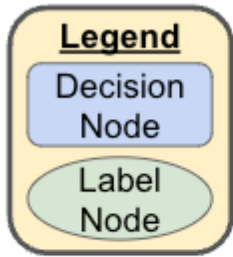


Overall Root

null

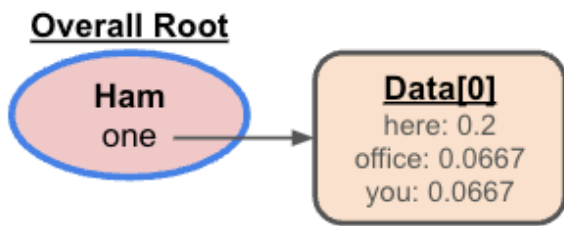
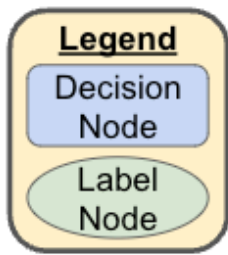
<u>Data</u>	<u>Data [0]</u> here: 0.2 office: 0.0667 you: 0.0667	<u>Data [1]</u> here: 0.0 office: 0.0667 you: 0.0667	<u>Data [2]</u> here: 0.0 office: 0.0667 you: 0.0667	...
<u>Labels</u>	<u>Labels [0]</u> Ham	<u>Labels [1]</u> Spam	<u>Labels [2]</u> Spam	...

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



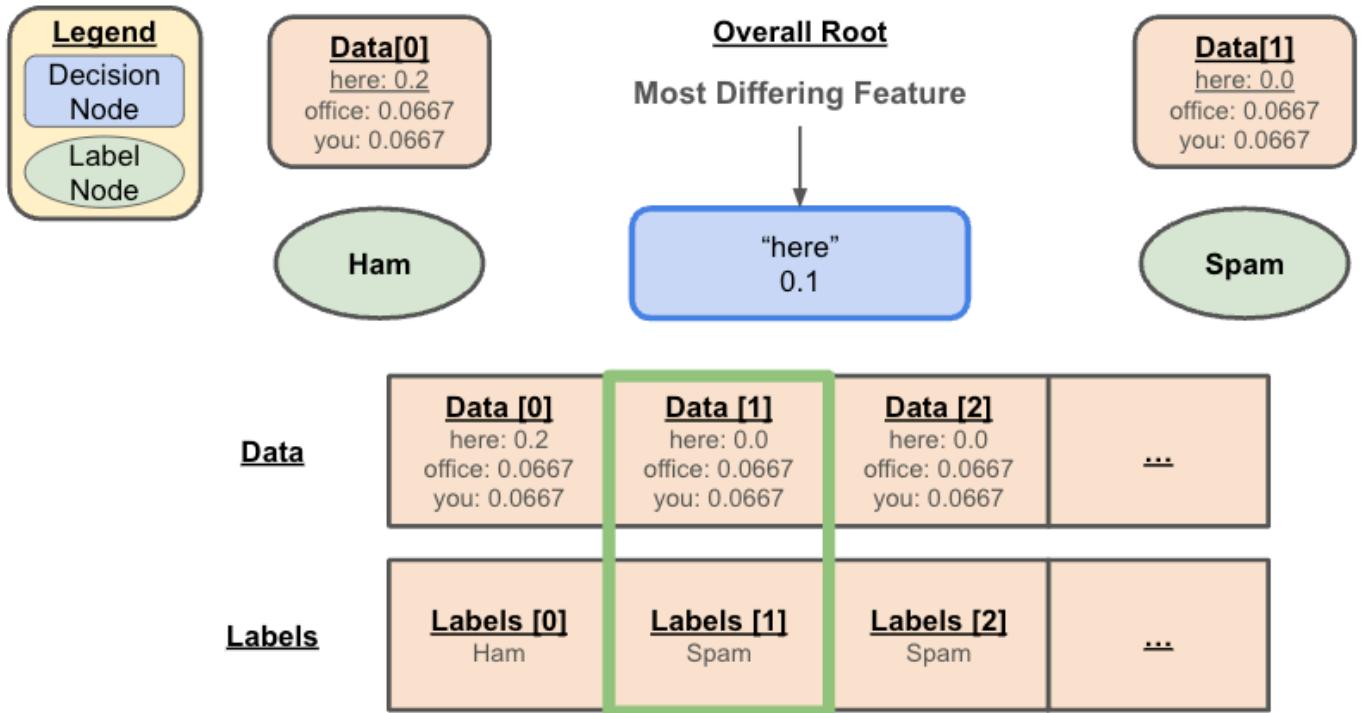
Data	<u>Data [0]</u> here: 0.2 office: 0.0667 you: 0.0667	<u>Data [1]</u> here: 0.0 office: 0.0667 you: 0.0667	<u>Data [2]</u> here: 0.0 office: 0.0667 you: 0.0667	...
Labels	<u>Labels [0]</u> Ham	<u>Labels [1]</u> Spam	<u>Labels [2]</u> Spam	...

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

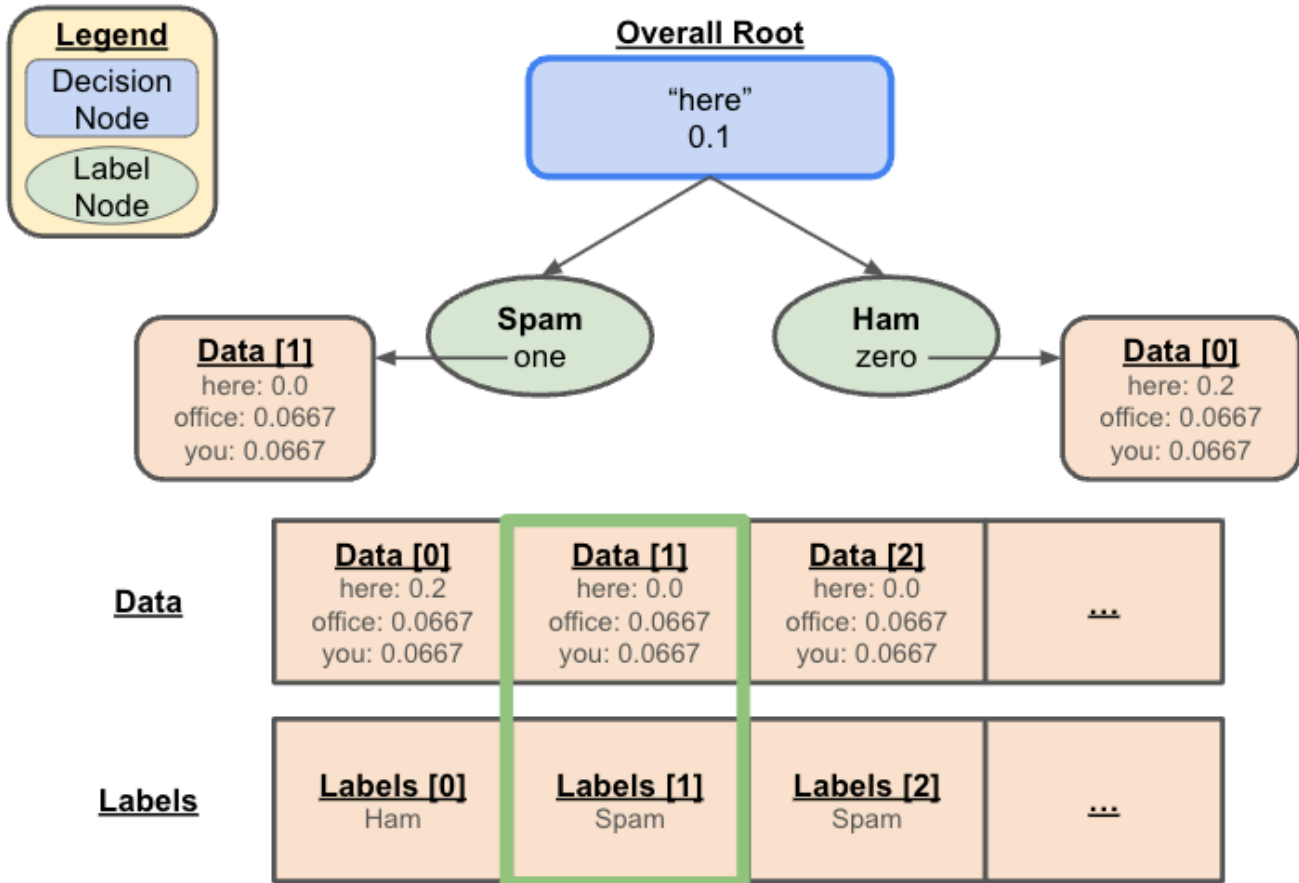


Data	Data [0] here: 0.2 office: 0.0667 you: 0.0667	Data [1] here: 0.0 office: 0.0667 you: 0.0667	Data [2] here: 0.0 office: 0.0667 you: 0.0667	...
Labels	Labels [0] Ham	Labels [1] Spam	Labels [2] Spam	...

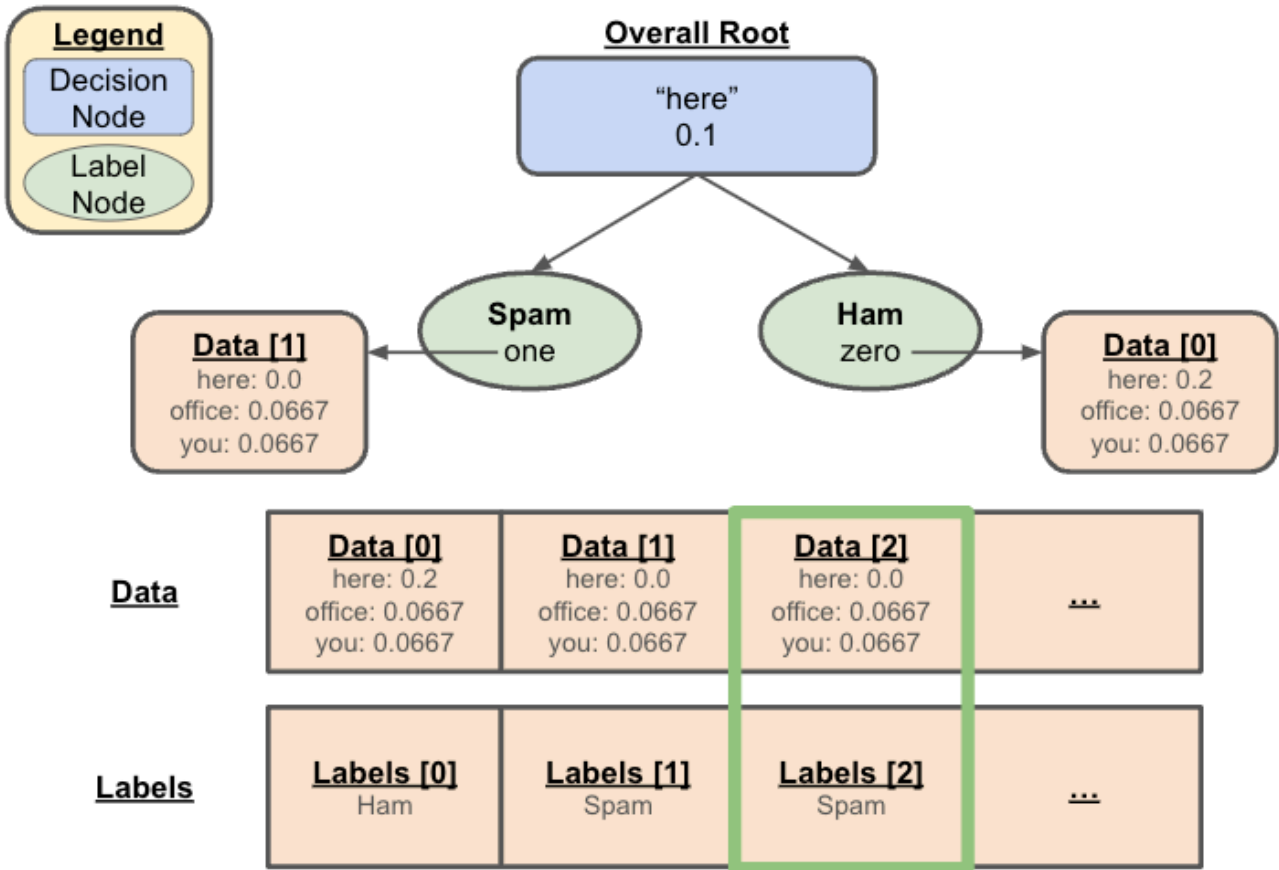
Then we see if the resulting label is correct. Our expected result is "Spam", but the one predicted by our model is "Ham". This is incorrect, so we need to create a decision node with the most differing feature between the two `TextBlock` objects (one previously stored in the Ham node, and the other from `Data`) based on their `get()` values:



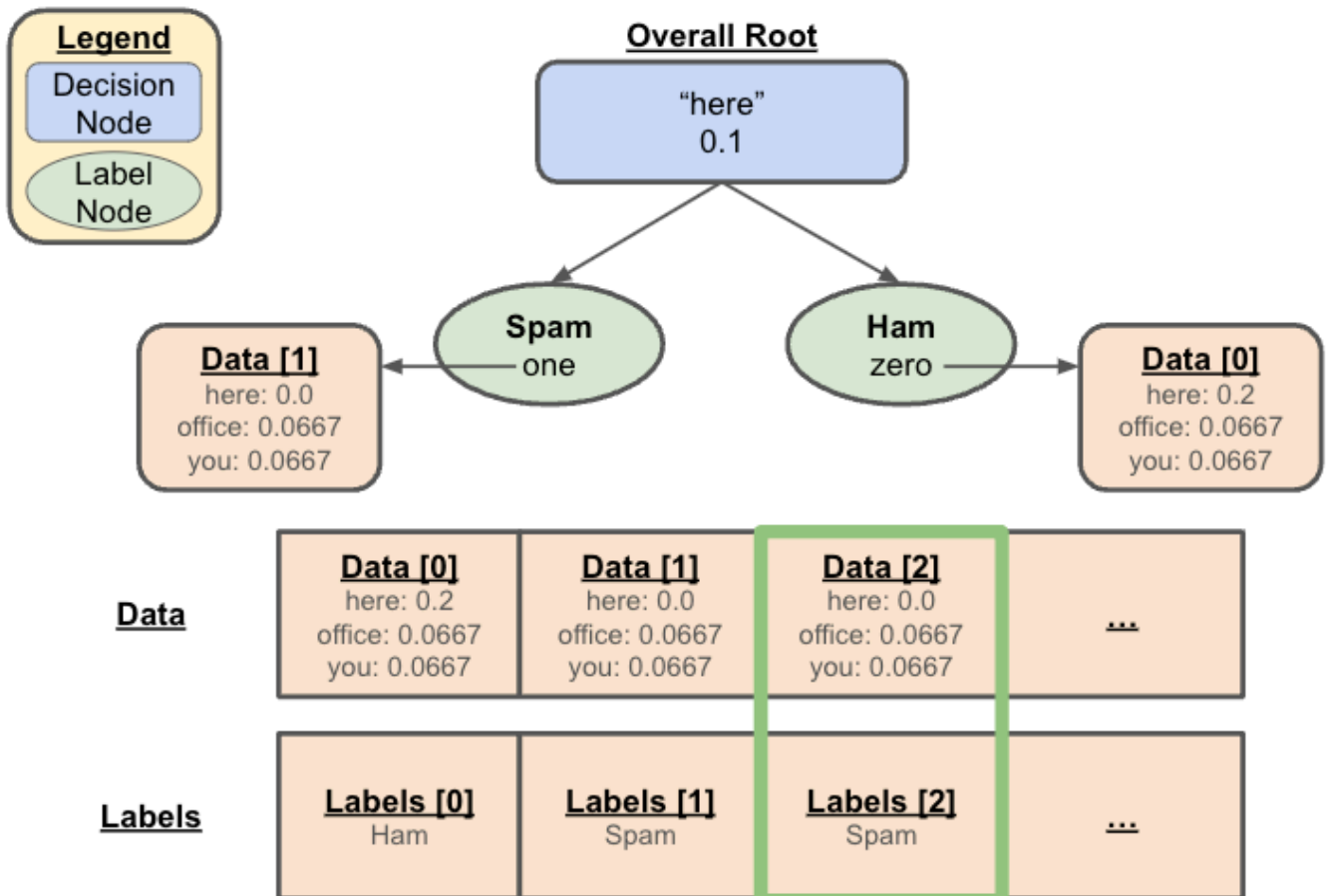
We can then utilize the provided methods to produce a new intermediary node that will allow us to correctly distinguish `Data[0]` vs. `Data[1]` using a feature and a threshold based on the algorithm described in this method's specification. All that's left to do is organize the label nodes appropriately as seen below:



Next, we continue to process the next data point at index 2. We'll repeat the algorithm as described above. First, traverse through the existing tree until we reach a leaf node:



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



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

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

```
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 our Email example above: 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`.

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

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

```
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.

```
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 initialize the values for *all* `final` fields in your node class — even if you do not plan to utilize the value.

File Format

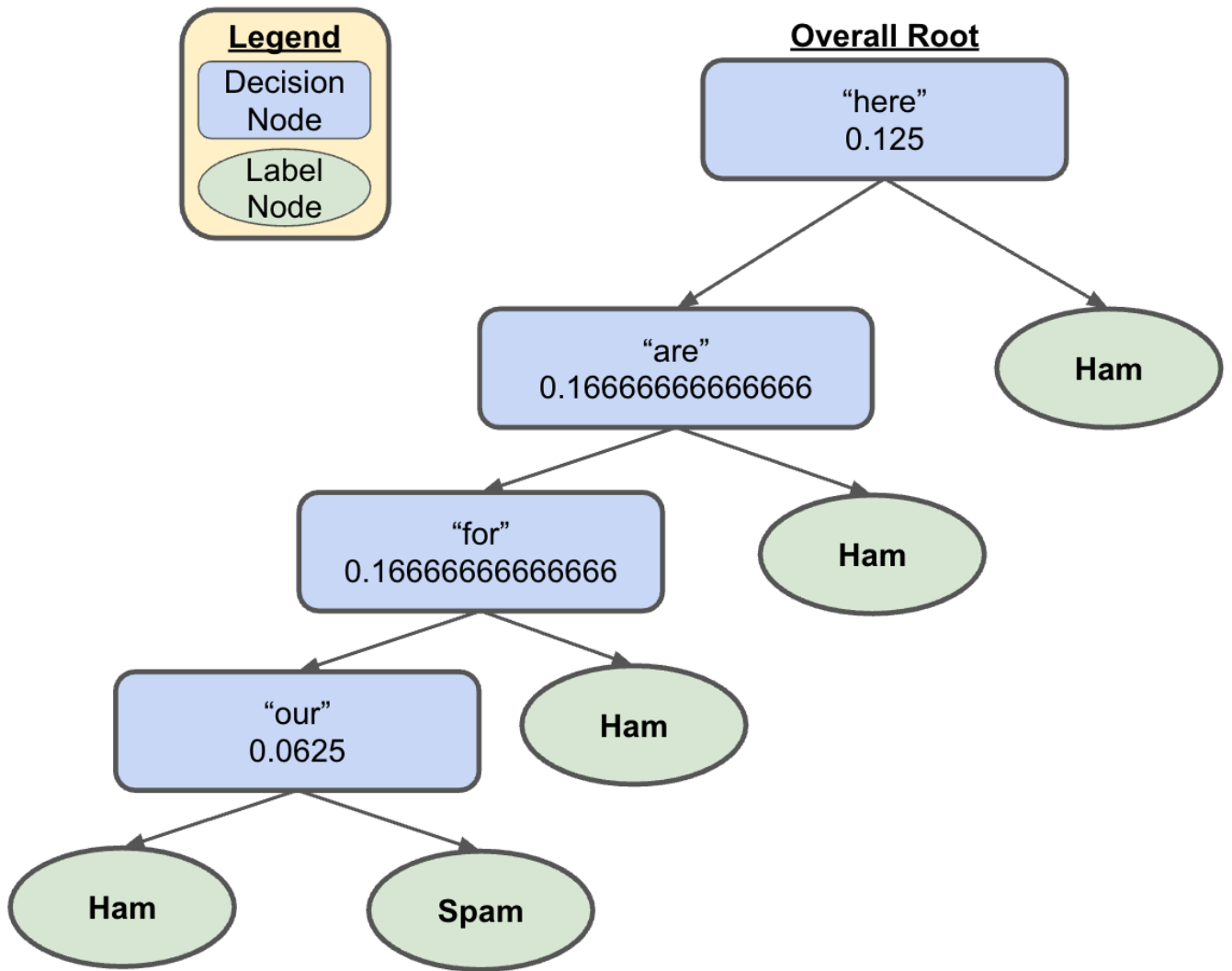
The files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent intermediary nodes and a single line to represent leaf nodes in the `Classifier`. The first line in each intermediary node pair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

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

▼ Expand

```
Feature: 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:



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).

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** and additional information is *italicized*).

1. This client visualization uses your Scanner constructor, `calculateAccuracy()`, and `classify()`. The constructor loads a pre-trained model from a given text file. The following inputs allow us to test its accuracy using the pre-set `TEST_FILE` (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! To begin, enter your desired mode of operation:
```

- 1) Train classification model
- 2) Load model from file

```
Enter your choice here: 2 // the Scanner constructor
```

```
Please enter the path to the file you'd like to load: trees/simple.txt
```

```
What would you like to do with your model?
```

- 1) Test with an input file
- 2) Get testing accuracy
- 3) Save to a file
- 4) Quit

```
Enter your choice here: 2 // calculateAccuracy()
```

```
Overall: 0.8637632607481853
```

```
ham: 0.9961365099806826
```

```
spam: 0.0
```

- 1) Test with an input file
- 2) Get testing accuracy
- 3) Save to a file
- 4) Quit

```
Enter your choice here: 1 // classify() called on every data point
```

```
Please enter the file you'd like to test: data/emails/test.csv
```

```
Results: [ham, 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, ham, h  
...
```

```
ham, 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
- 2) Get testing accuracy
- 3) Save to a file
- 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! To begin, enter your desired mode of operation:
```

- 1) Train classification model
- 2) Load model from file

```
Enter your choice here: 1 // Classifier(List<TextBlock>, List<String>) constructor
```

```
What would you like to do with your model?
```

- 1) Test with an input file
- 2) Get testing accuracy
- 3) Save to a file
- 4) Quit

```
Enter your choice here: 2 // calculateAccuracy()
```

```
"MADISON, HAMILTON": 1.0
```

```
Overall: 0.9230769230769231
```

```
MADISON: 1.0
```

```
JAY: 1.0
```

```
HAMILTON: 0.8823529411764706
```

- 1) Test with an input file
- 2) Get testing accuracy
- 3) Save to a file
- 4) Quit

```
Enter your choice here: 3 // save()
```

```
Please enter the file name you'd like to save to: destinationFile.txt
```

- 1) Test with an input file
- 2) Get testing accuracy
- 3) Save to a file
- 4) Quit

```
Enter your choice here: 4
```



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



NOTE: You may notice that your numbers may differ between each run. That is totally fine! That's because the `DataLoader` class shuffles the rows around so that the trees don't always have the same structure (because that would be boring). If you wish to test your `Classifier(List<TextBlock>, List<String>)` constructor consistently, feel free to comment out line 27 in the `DataLoader` class.

Testing

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

□ Implementation Guidelines

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

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

Development Guide

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

1) Design your Node

First, design your node class that represents both the intermediary 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, intermediary 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!

2). 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** (described in the [Specification](#) slide):

These files will contain a pair of lines to represent intermediary nodes and a single line to represent leaf nodes in the `Classifier`. The first line in each intermediary node pair will start with "Feature: " followed by the feature and the second line will start with "Threshold: " followed by the threshold. Lines representing the leaf nodes will simply contain the label.

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

 The tests for your `save` implementation are tied to a working `Scanner` constructor 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.

3). 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()`.

Remember that this file is stored in pre-order format, where the feature and threshold for intermediary 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` if `input` is null.



HINT: 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?**



At this point, test your `Scanner` constructor and `save` implementations. Feel free to write the method signature of later methods so that the tests compile, but we don't recommend moving forward in this assignment until these two methods are passing the provided tests.

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` if `input` is null



At this point, test your current implementation. Feel free to write the method signature of later methods so that the tests compile, but you we don't recommend not moving forward in this assignment until the `classify` method is passing

5). Two List Constructor

Here is where we actually "train" our model, and 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 `results` is null

- `data` and `results` are not the same size
- `data` or `results` is empty

Next, your implementation should follow the following algorithmic approach (copied from the spec — diagrams depicted there):

- 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 `results.get(<index>)`.
- For a specific index, traverse through the current classification tree (the tree you are currently building) until you reach a leaf node.
 - If the node's label matches the current label, do nothing (our model is accurate up to this point).
 - If the label is incorrect, create a new decision node between the data used to create the original leaf node* and our current input.
 - You should use the `findBiggestDifference` method within the `TextBlock` class to find the feature and generate the new midpoint as the threshold.
 - Note that there is no difference between calling `a.findBiggestDifference(b)` and `b.findBiggestDifference(a)`. Both will return the same string.
 - After the new feature node is constructed, the original leaf node and current input should be placed appropriately



* 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: 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!!! ☐ Make sure your code adheres to the [Code Quality Guide](#) and [Commenting Guide](#)!