Programming Assignment 4: Spam Classifier

Background

The rise of "Machine Learning" and "Artificial Intelligence" has been hard to ignore in the past decade. While the more well-known applications include ChatGPT and Dall-E, there are a number of other uses for these powerful algorithms. These include assisting biologists in the drug discovery process, helping medical professionals diagnose diseases early, and predicting erratic wildfire movements to save lives.

In essence, "Machine Learning" and "Artificial Intelligence" are subsets of Computer Science concerned with using trends from previous examples to predict things about unseen data. Yet, it is important to remember that these algorithms aren't magic with limitless potential – they simply guess the most likely outcome based on many, many previous examples. This means that any algorithm's predictions are only as good as the data it was built upon, which can easily be biased in some way. Thus, it is important to recognize and advocate for appropriate uses of these models, regardless of how miraculous they seem.

Terminology

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

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

classifier. In our tree, if the current input is less than the threshold we should go left. If it's greater than or equal to, we should go right.

Specification

Learning Objectives

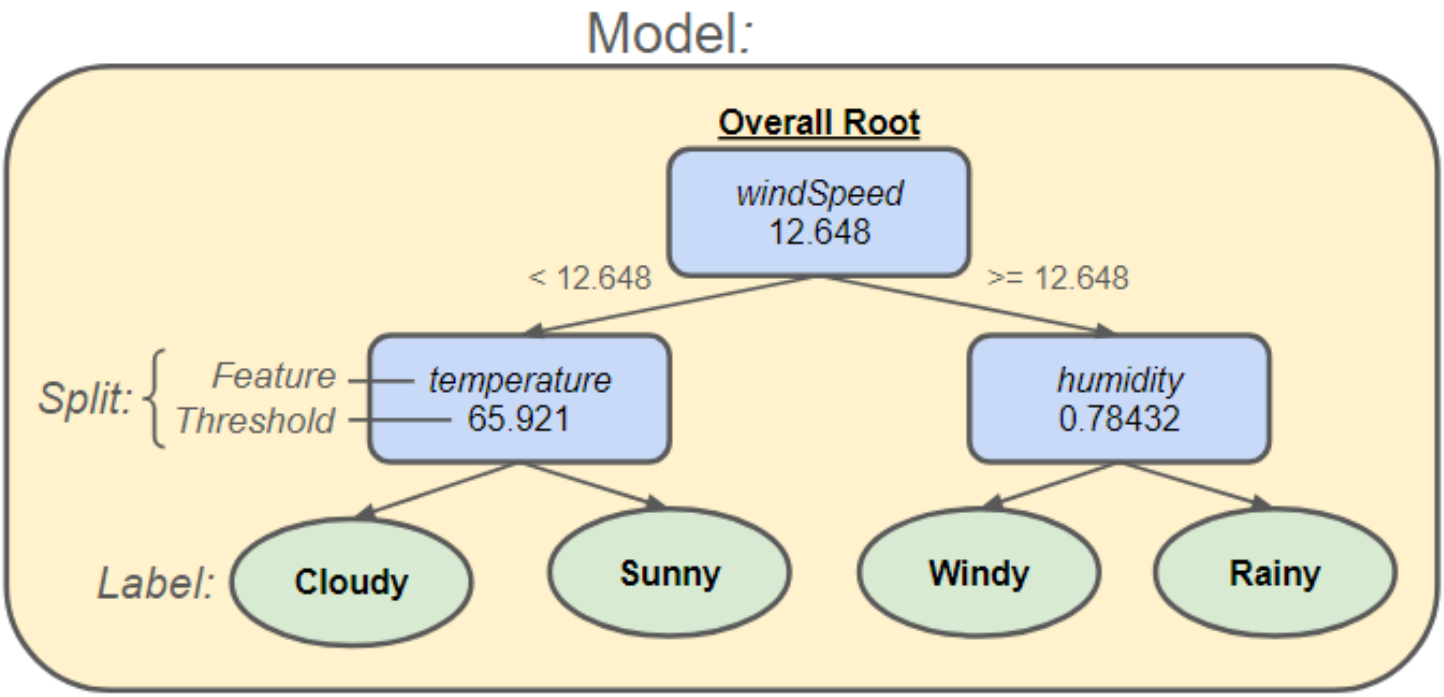
By completing this assignment, students will demonstrate their ability to:

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

Assignment

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

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



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

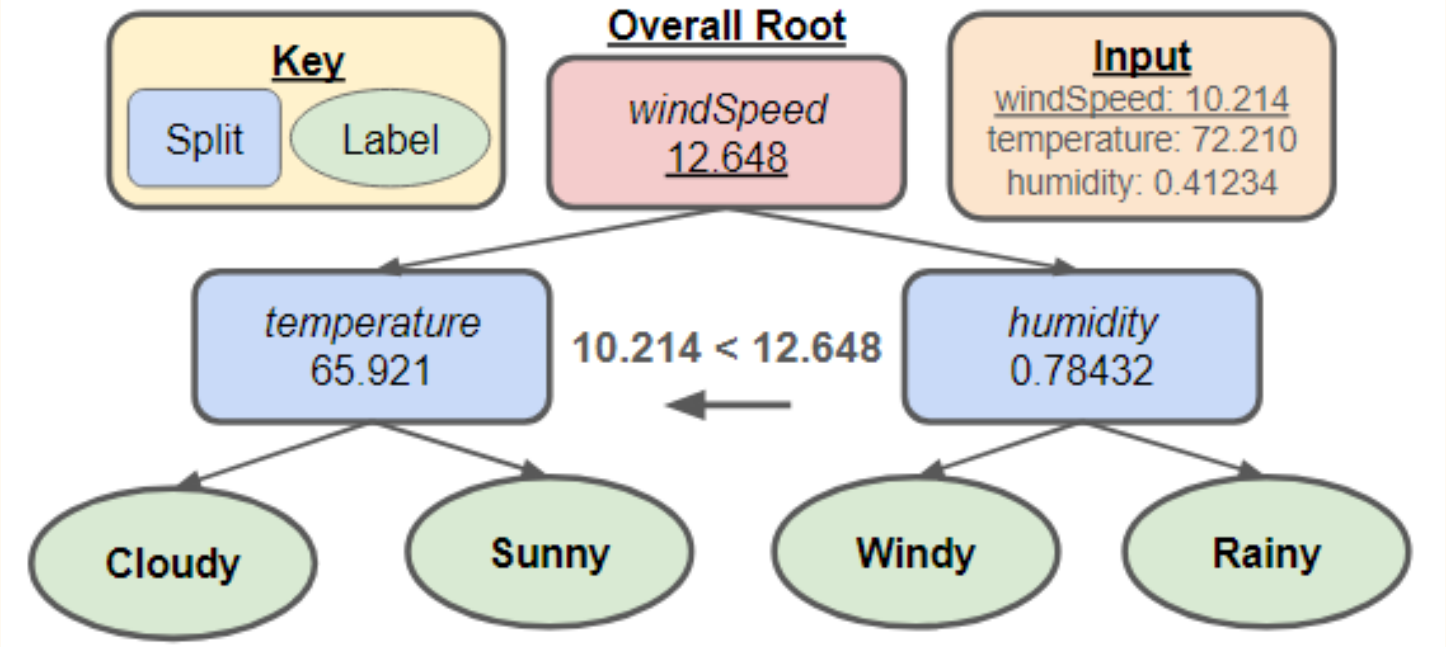
Below we'll trace through a sample input with our example weather model.

▼ Expand

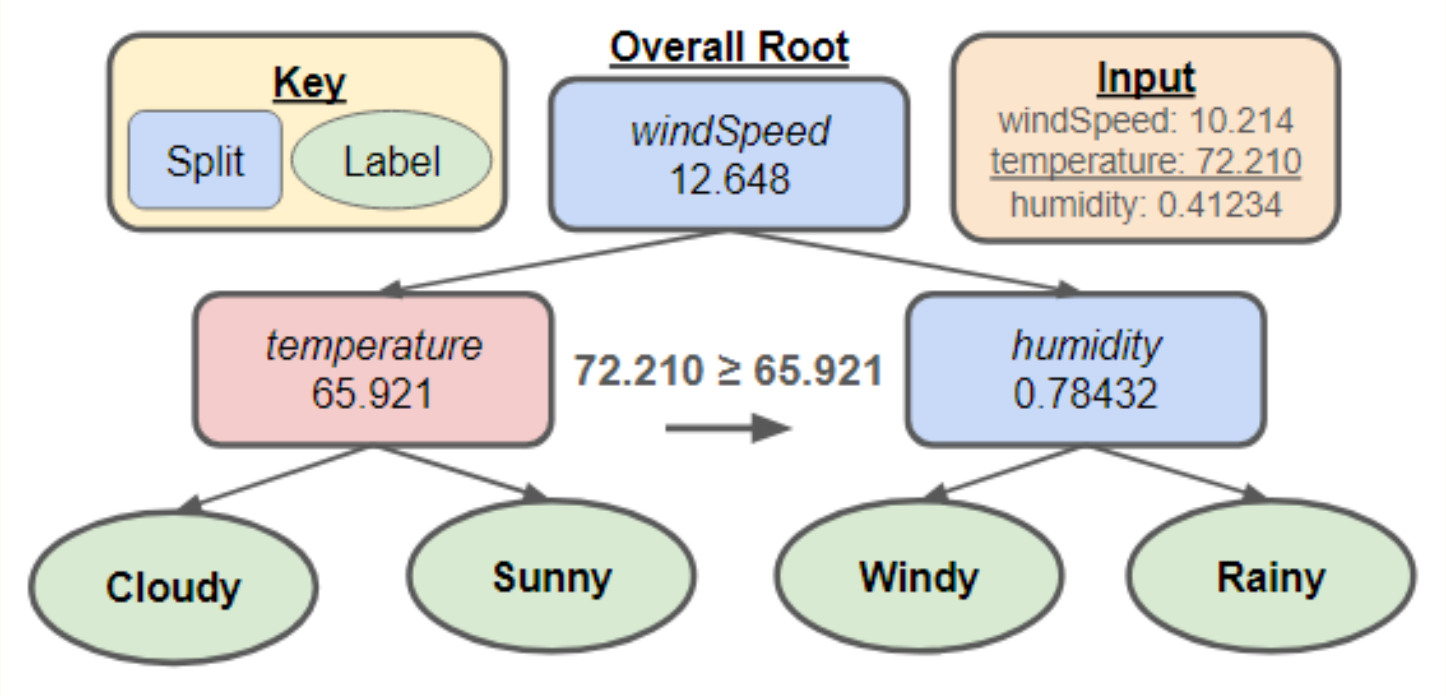
We'll begin at the root node with the following input:

```
       windSpeed, temperature, humidity
input:    10.214,      72.210,  0.41234
```

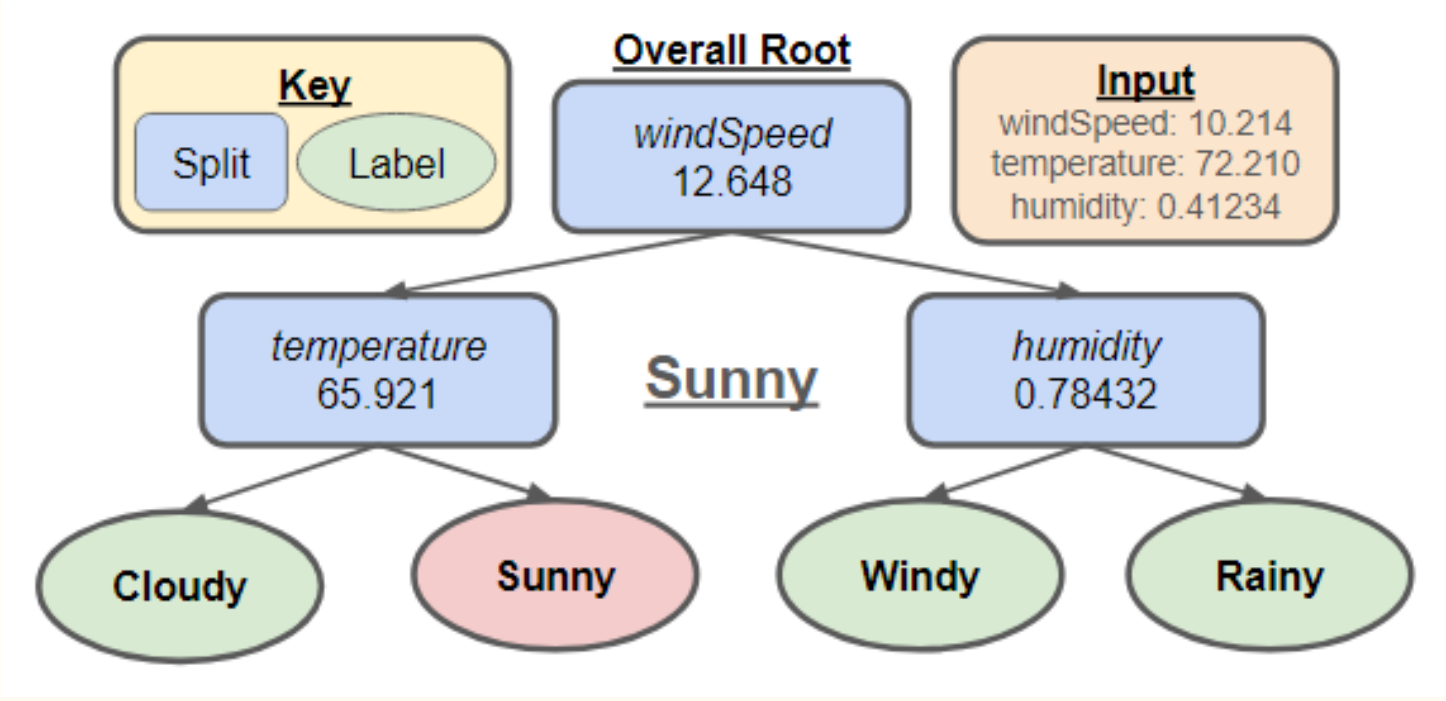
1. Since the windSpeed feature of the input is < the threshold (12.648) we'll travel left to the temperature node



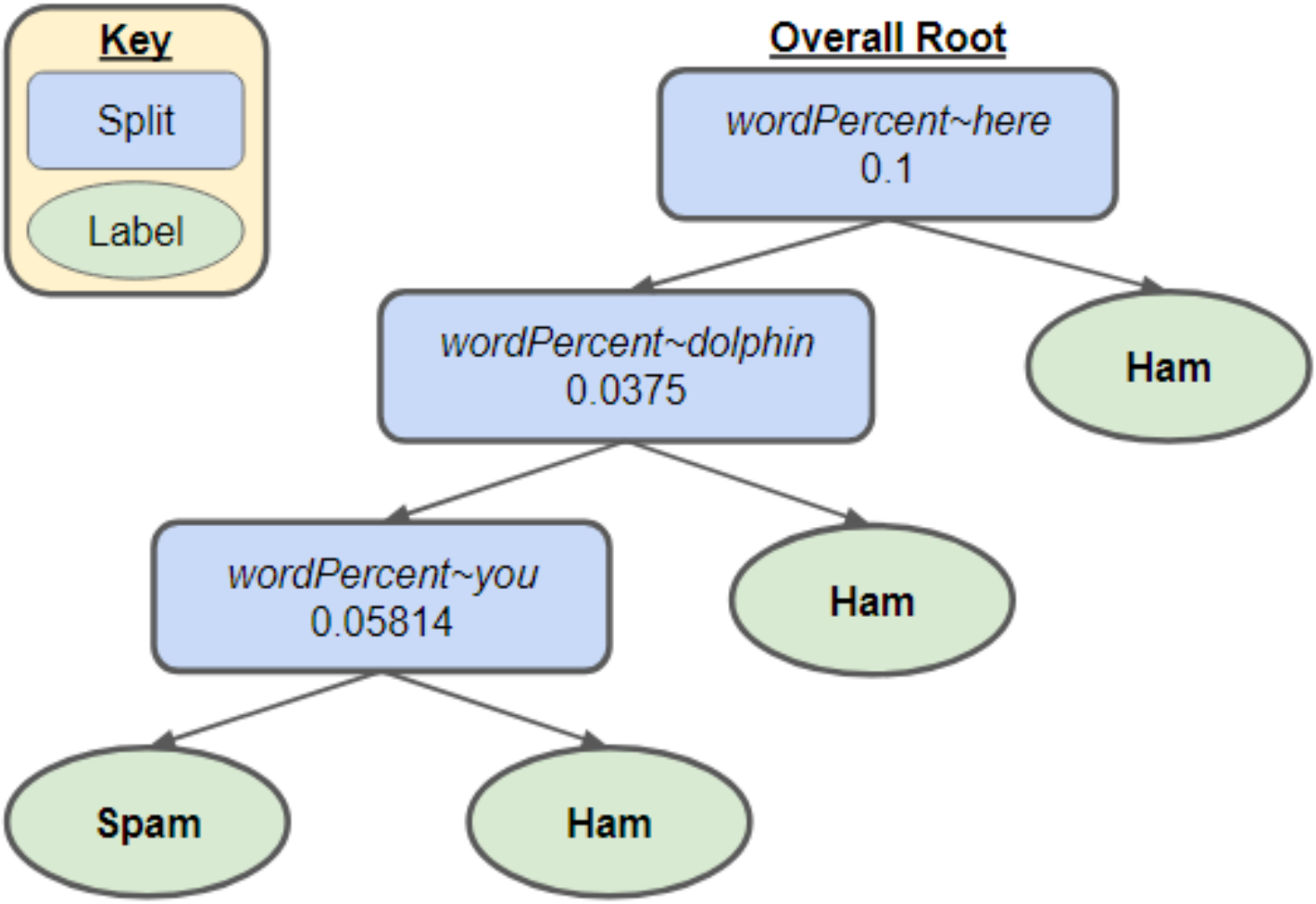
2. Since the temperature feature of the input is >= the threshold (65.921) we'll travel right to the Sunny node



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



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



Similar to the above, you'll notice that the leaf nodes of this tree represent labels ("Spam" or "Ham" – a funny way of writing not spam) while the intermediary values represent a split on some feature of our data (wordPercent). Notice that the features in this example are slightly different from the

weather one above. Specifically, wordPercent is the only feature within this model; however, we also need to track the specific word we're comparing the percentage of. This is accomplished by appending the word preceded by some arbitrary "splitter" character (in this case '~') that separates the two.

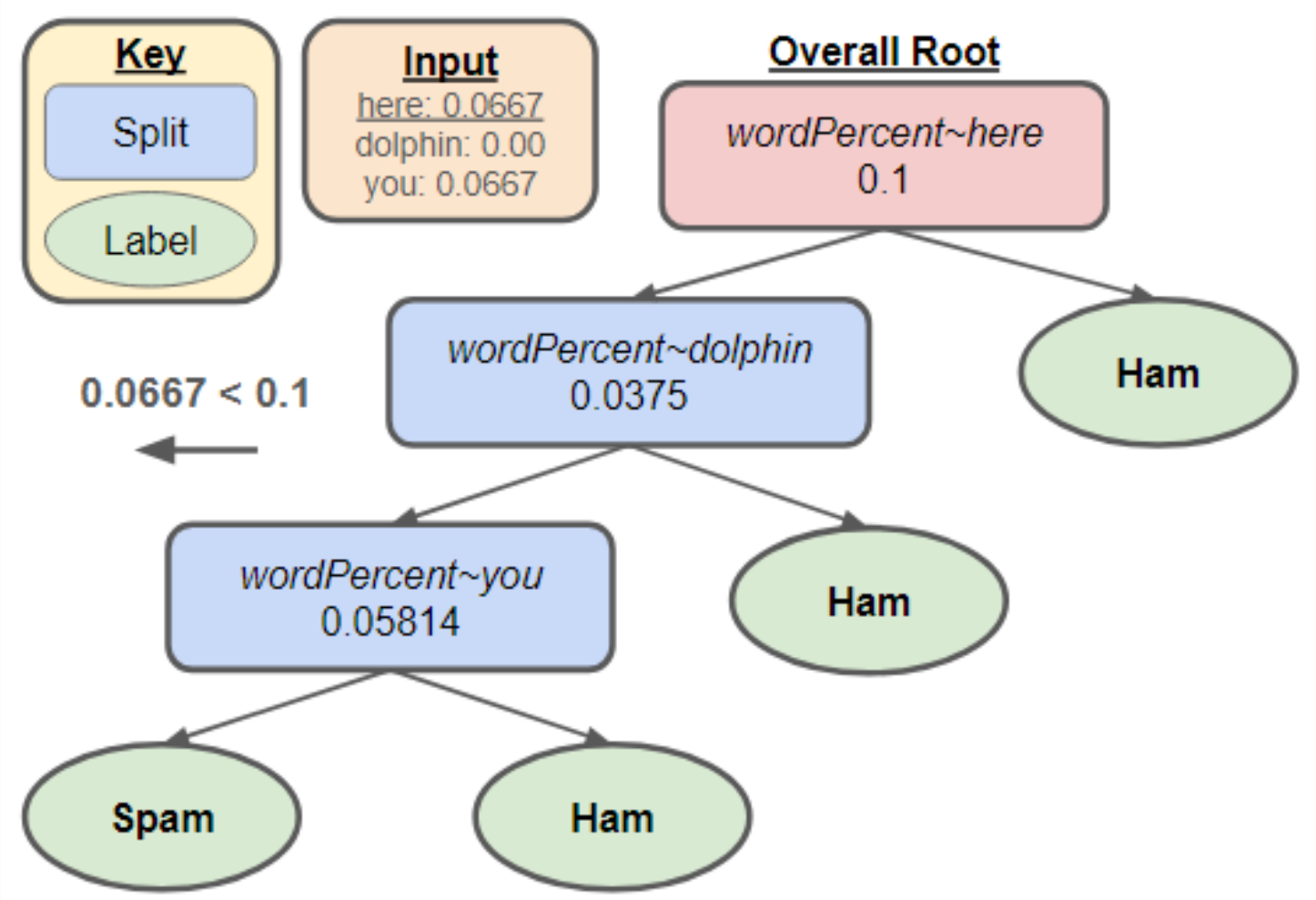
To solidify this idea, we'll trace through an input much like the weather example above.

▼ Expand

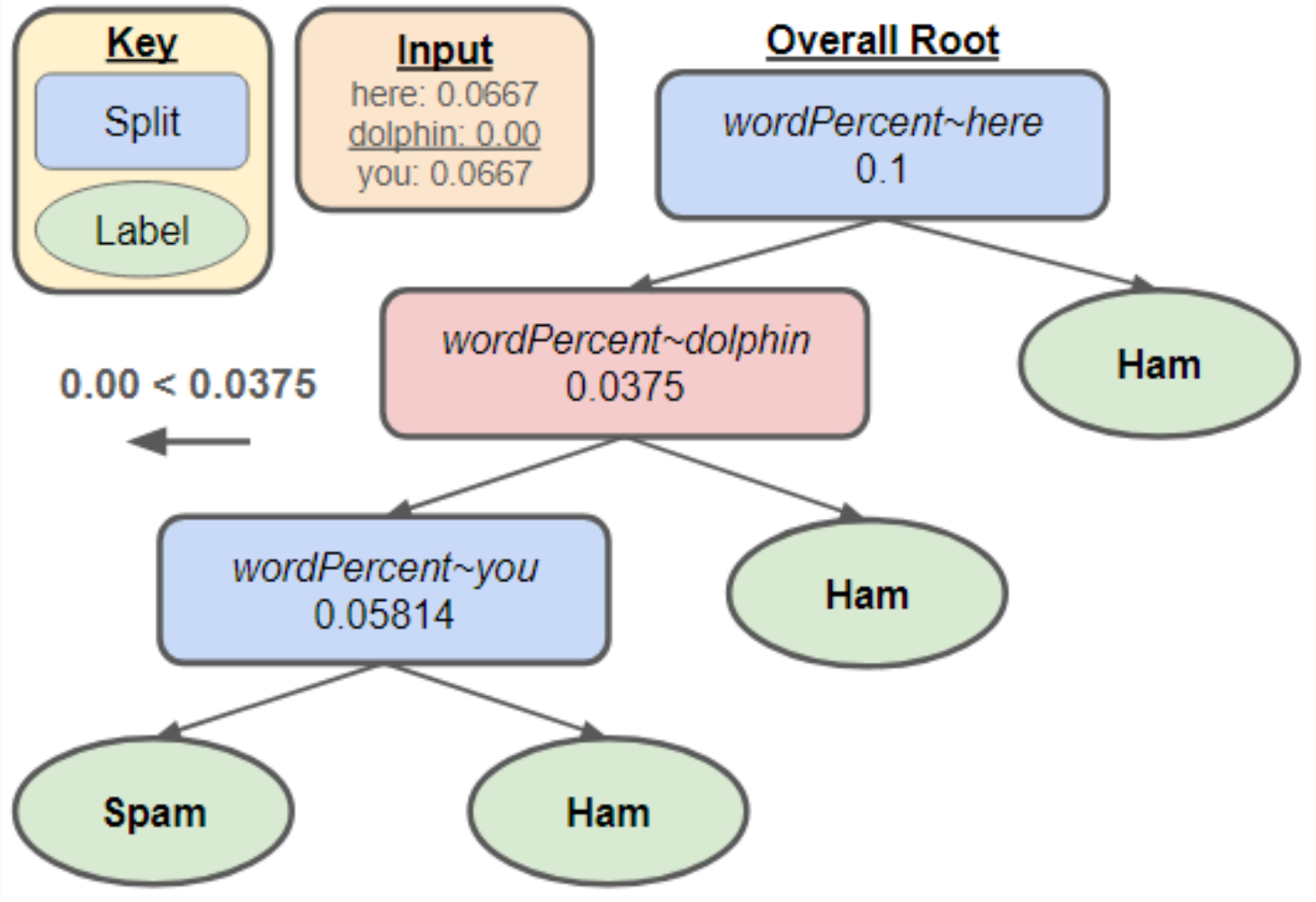
We'll begin at the root node with the following input:

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

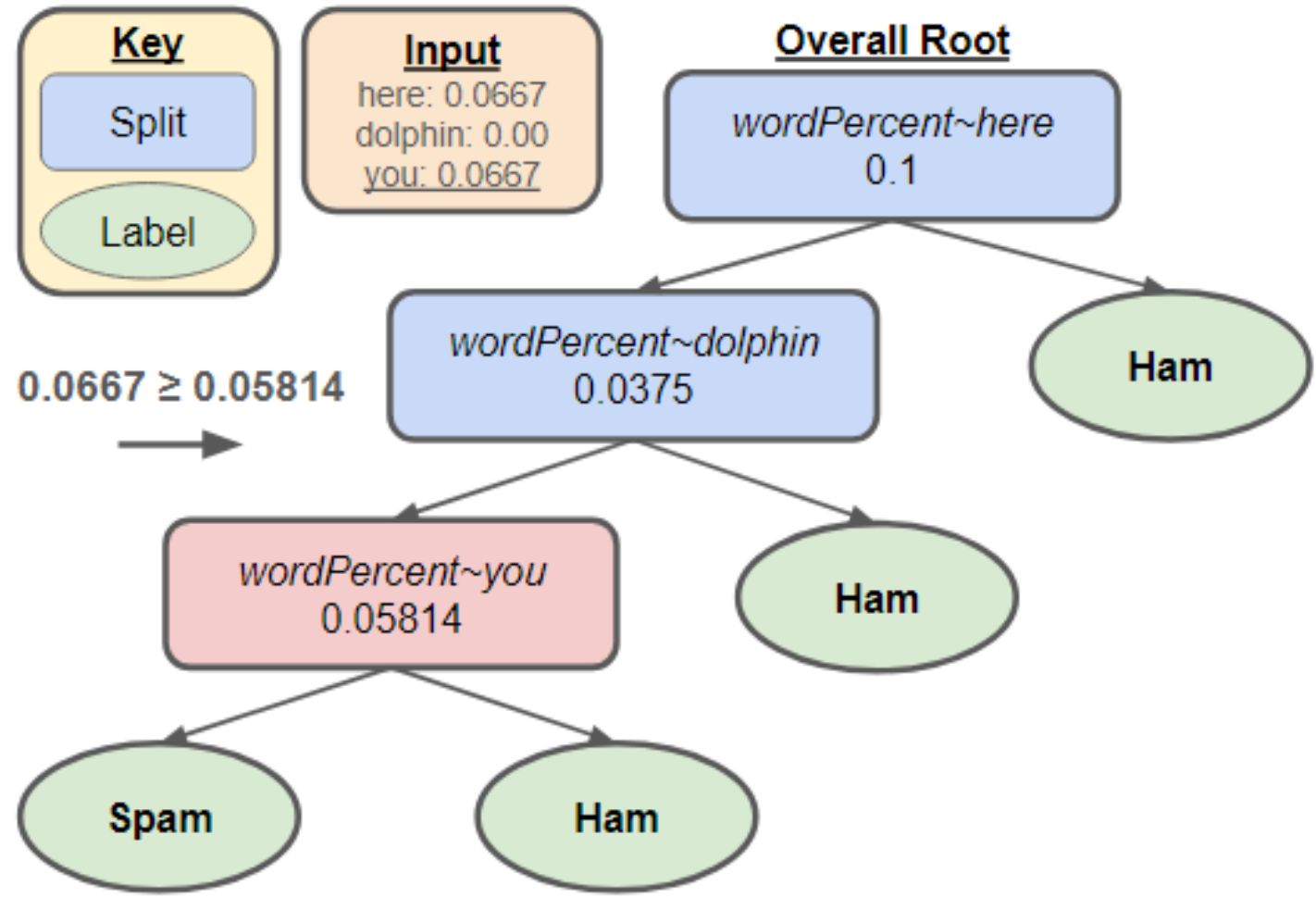
1. Since 'here' consists of 6.67% of the input email, which is < the threshold (10.00%) we'll travel left to the wordPercent~dolphin node



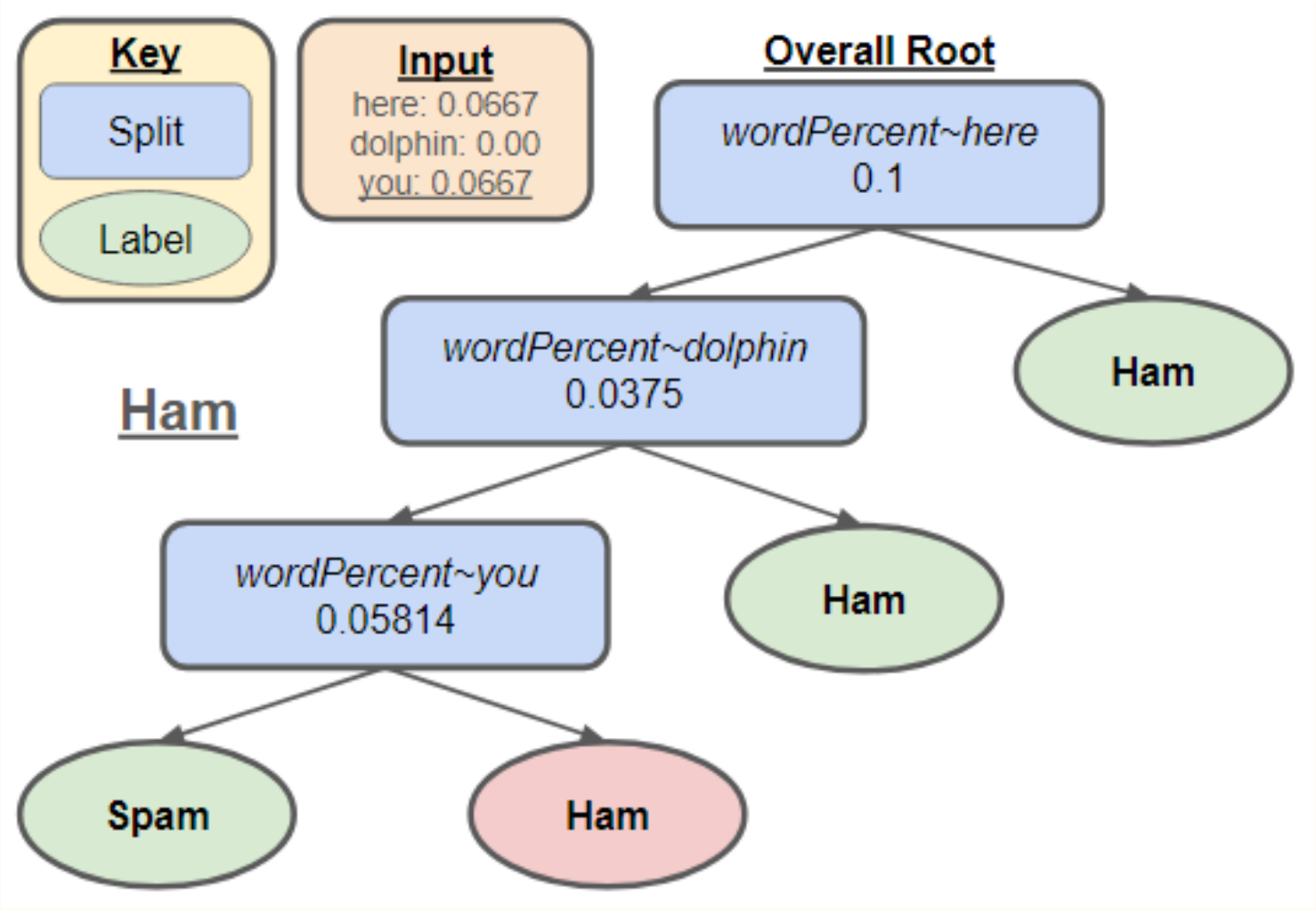
2. Since 'dolphin' consists of 0.00% of the input email, which is < the threshold (3.75%) we'll travel left to the wordPercent~you



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



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



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

System Structure

Classifiable.java

Any data point we want to classify via our model must implement this interface. It defines three methods:

▼ Expand

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

- Returns the corresponding value for the given feature.
    - Although there are classification trees where it would make sense to return something else (imagine a color feature within a real estate dataset), since our implementation is only dealing with thresholds this must return a double.

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

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

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

- Returns a partition (`Split`) between this data point and `other`.
    - How this is computed is up to the implementer (and is a large part of the complexity of our model).
    - Note that there is no difference between calling `one.partition(two)` and `two.partition(one)`. Both will return `Splits` with the same feature and threshold.

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

Classifier.java

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

▼ Expand

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

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

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

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

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

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

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

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

Split.java

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

▼ Expand

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

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

```
public String getFeature()
```

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

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

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

```
public String toString()
```

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

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

Required Operations

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

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

- Load the classification tree from a file connected to the given Scanner. You may assume that the `sc` is non-null and that the format of the input file matches that of the `save` method described within `Classifier`.
    - Importantly, in this method, you should only read data from the file using `nextLine` and convert it to the appropriate format using `Double.parseDouble`.
- This method should throw an `IllegalStateException` if the tree is empty after reading it in from this file

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

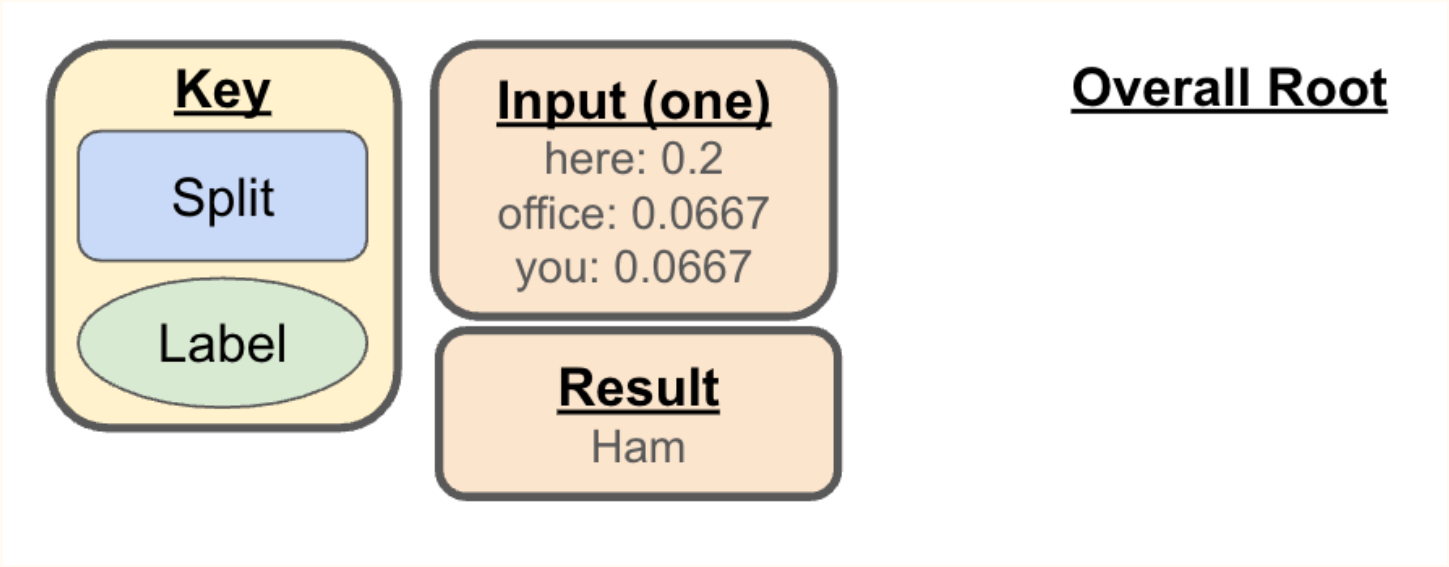
- Create a classification tree from the input data and corresponding labels.
    - **Note that you are building the tree up from scratch in this constructor**.
- The lists should be traversed in parallel, where the label corresponding to `data.get(i)` can be found at `results.get(i)`. The general construction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

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

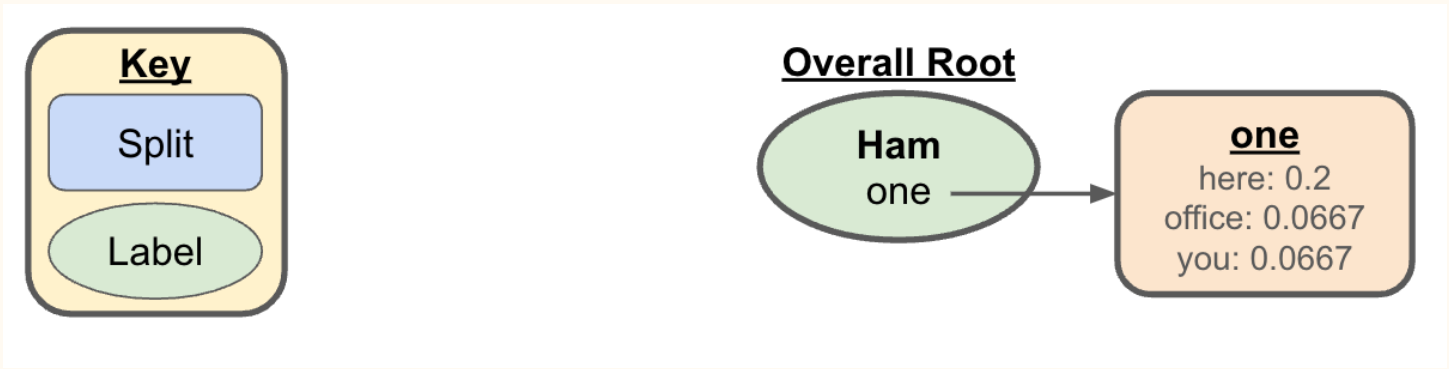
The algorithm above is further shown in the following diagrams:

▼ Expand

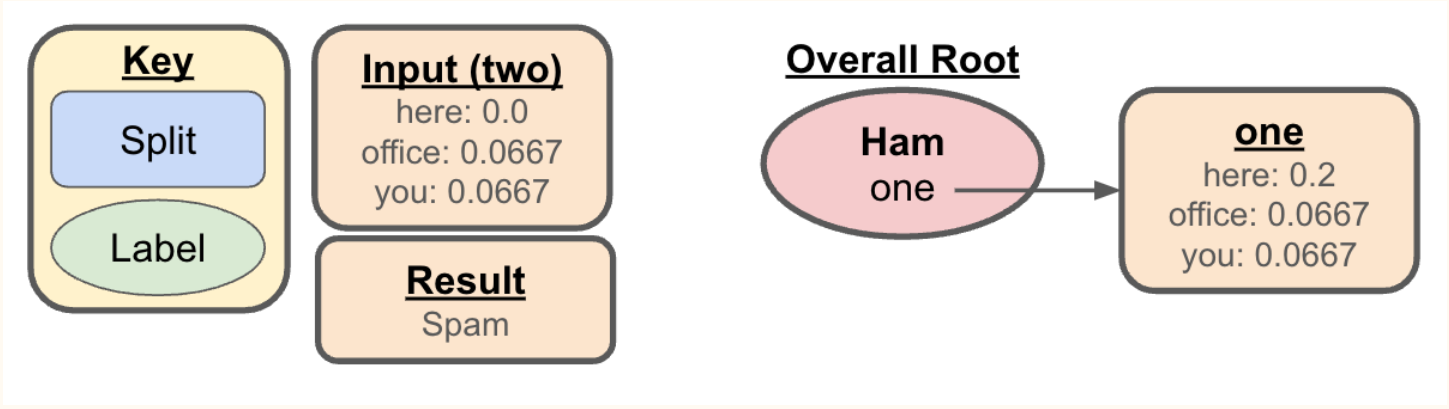
We start with an empty tree and process the first input:



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

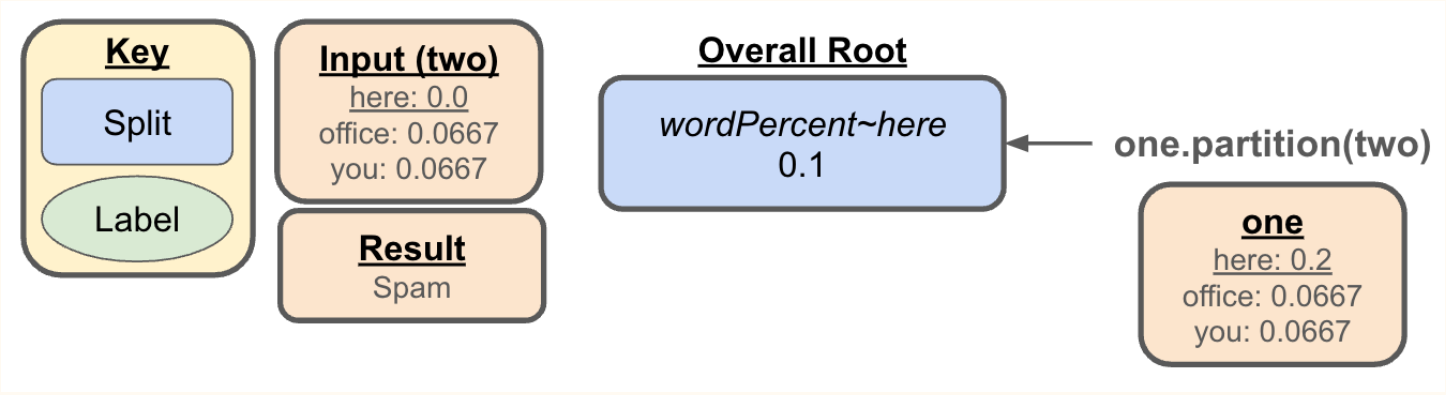


Note that this node also stores a reference to the data used to create it. This will be useful in the next step. Once we've processed the first data point, we move on to the second. We traverse through the existing tree until reaching a leaf node (which just so happens to be the only node in our tree):

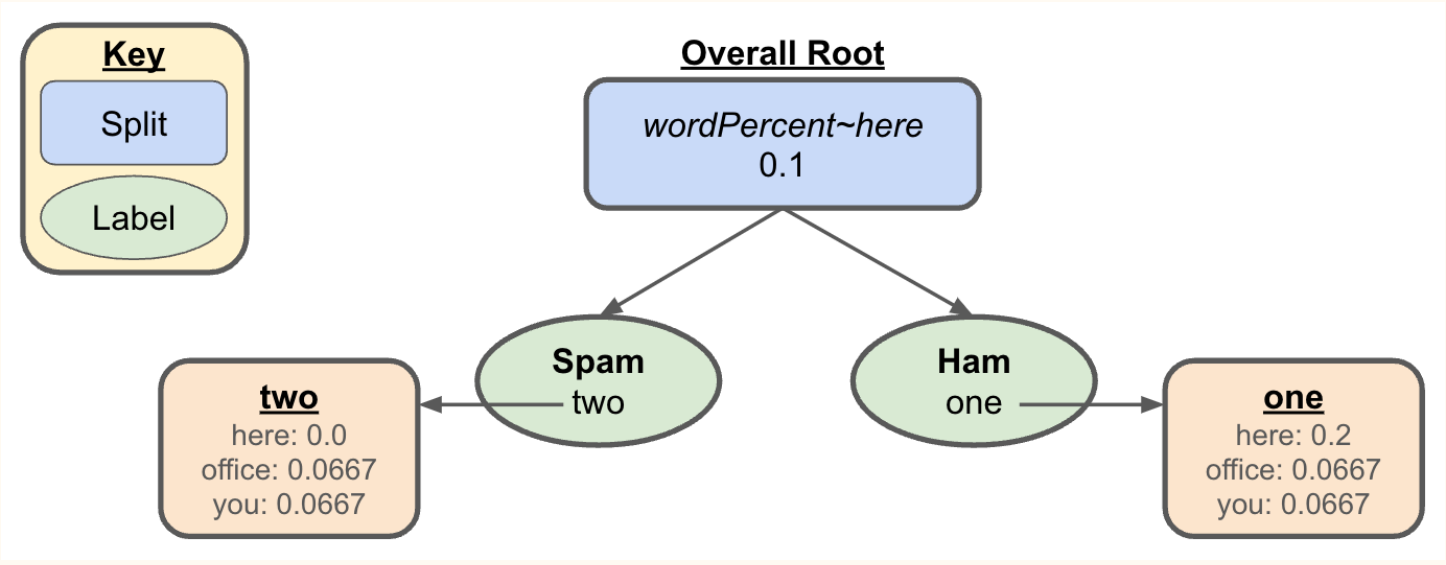


Then we see if the resulting label is correct. Our expected result is "Spam", but the one predicted by our model is "Ham". This is incorrect, so we need to create a new split via the

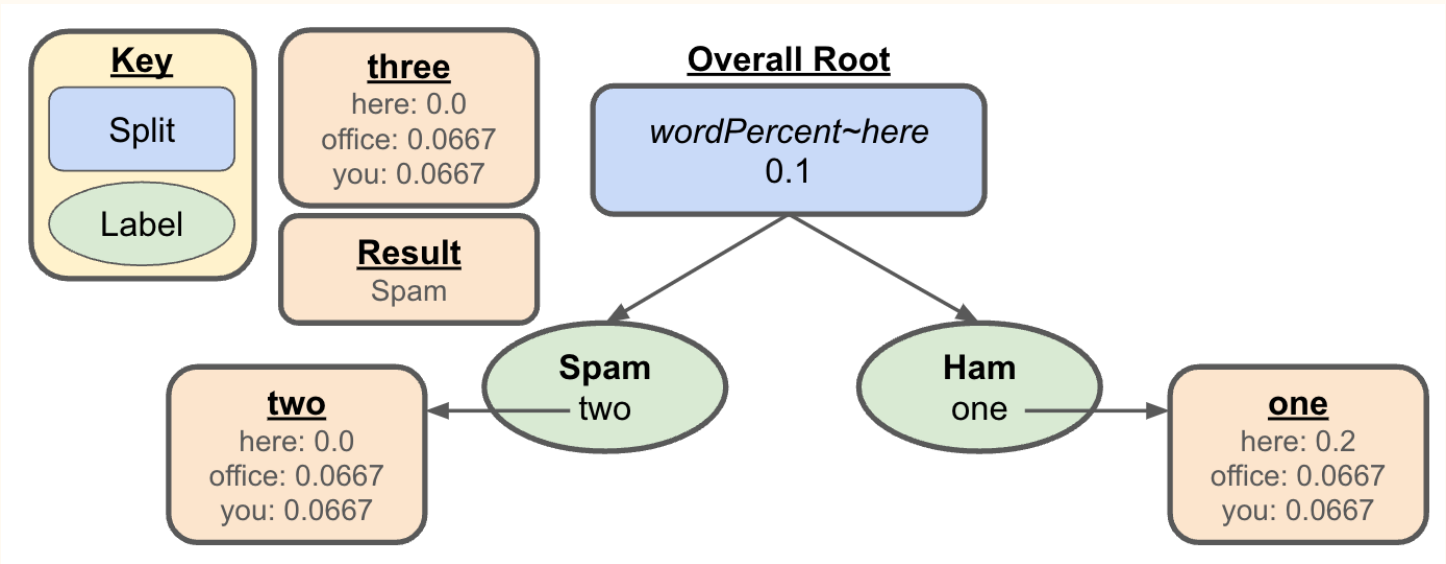
`Classifiable.partition()` method:



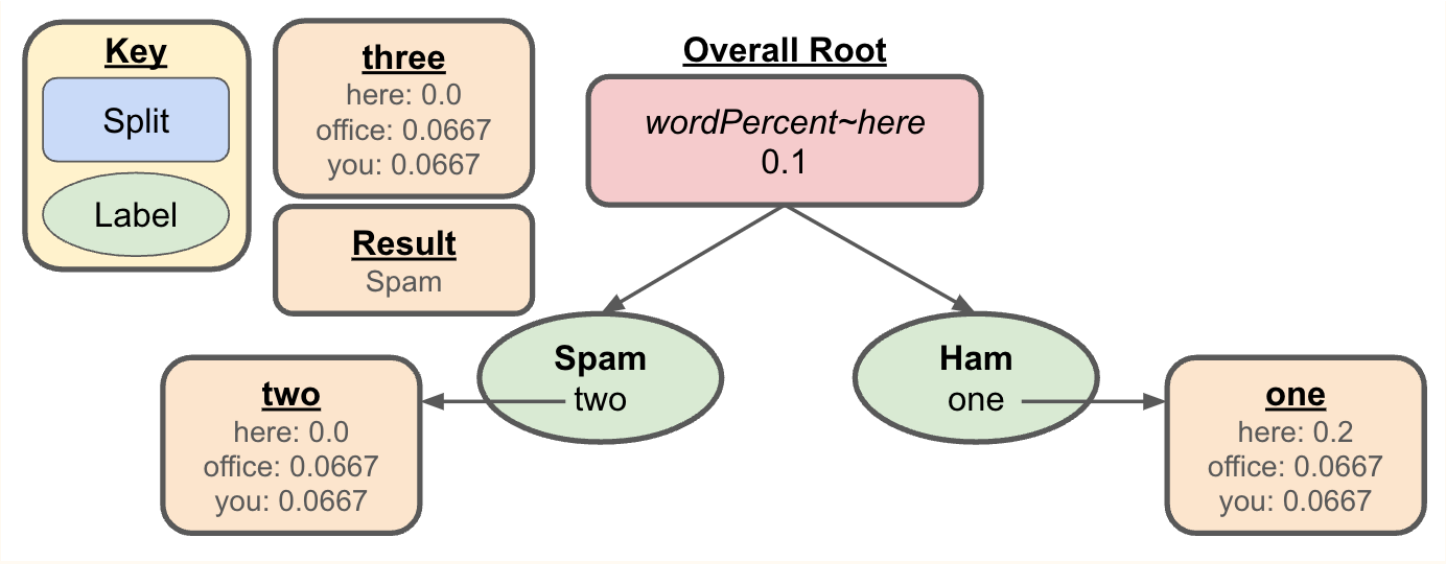
This will return a new `Split` which we can then store within a new intermediary node that will allow us to correctly distinguish `one` vs. `two`. All that's left to do is organize the label nodes appropriately as seen below:



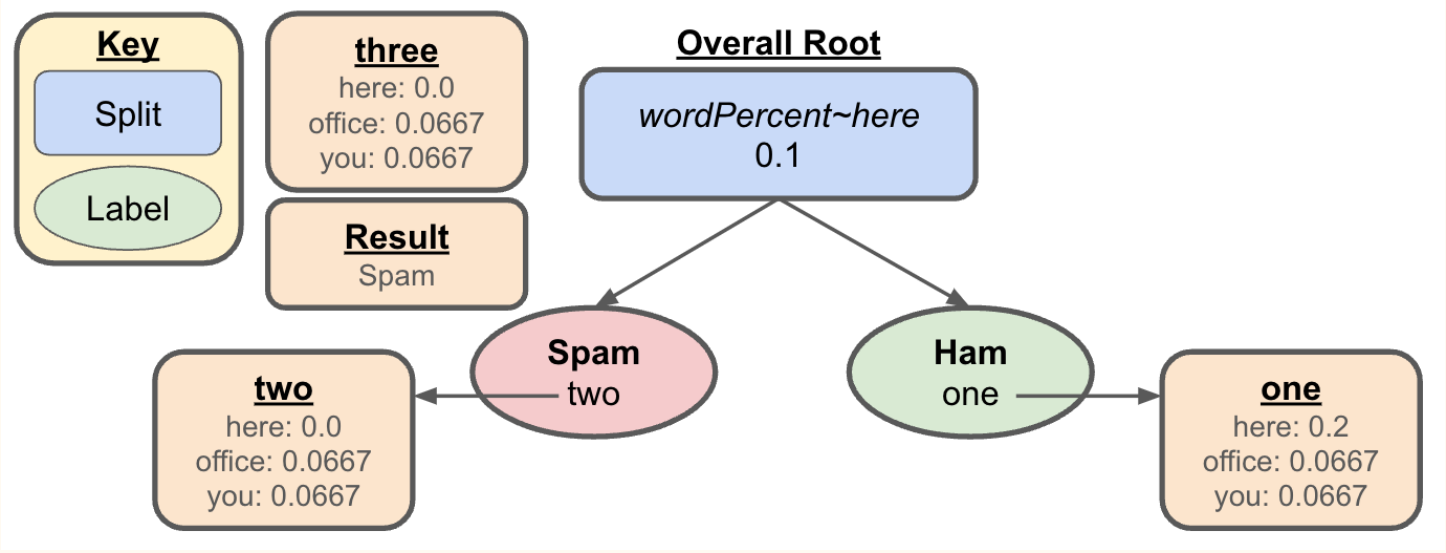
Furthermore, we can imagine undertaking this process with a third datapoint as depicted below



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



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



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

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

ClassificationNode

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

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

- The fields of the `ClassificationNode` class must be `public`.
    - All data fields should be declared **`final`** as well. This does not include fields referencing the children of a node.
- The `ClassificationNode` class must not contain any constructors or methods that are not used by the `ClassificationTree` class.

File Format

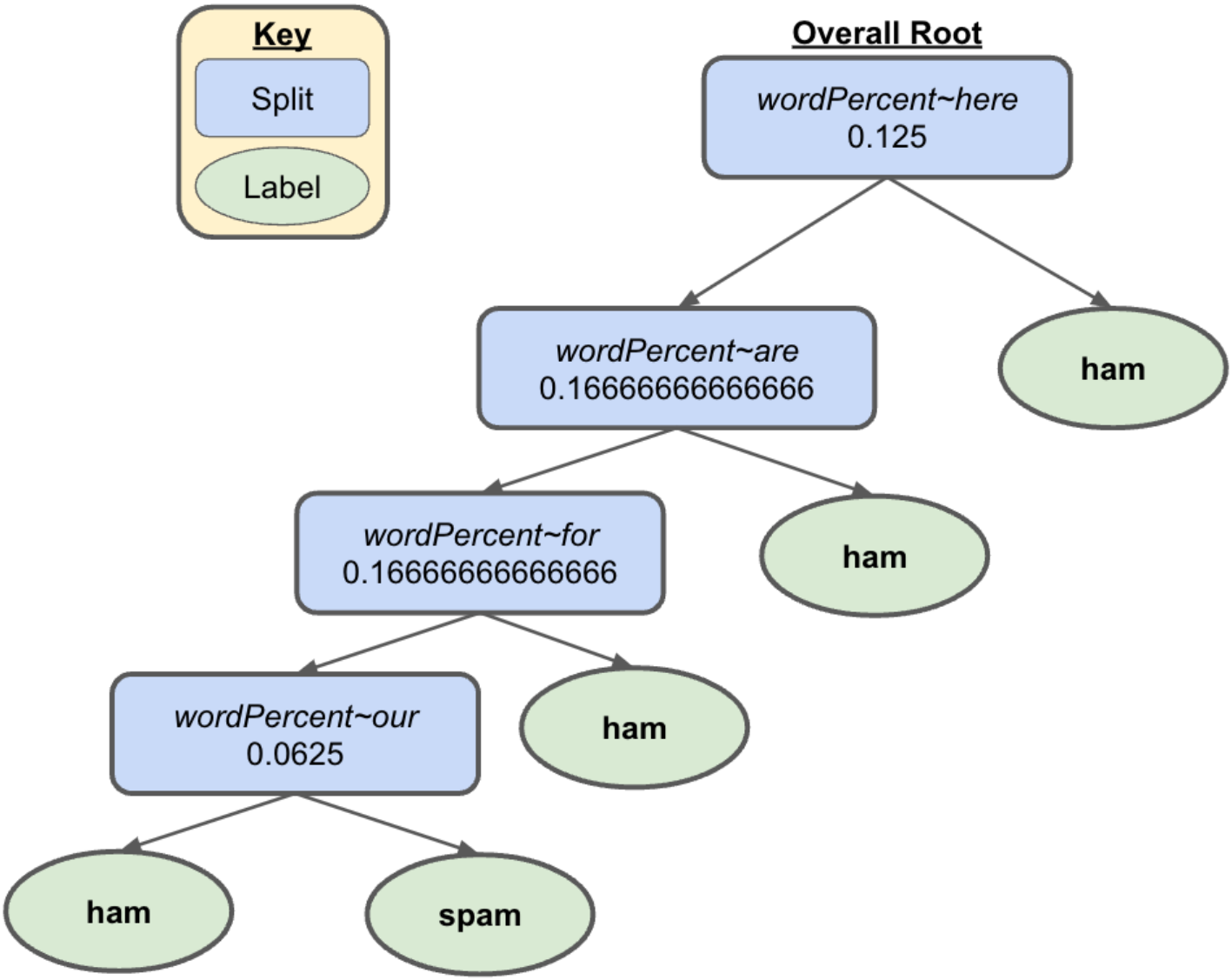
The files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent intermediary nodes and a single line to represent leaf nodes in the `ClassificationTree`. The first line in each intermediary node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

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

<details>
<summary>Expand</summary>

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

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



Try out your Classifier!

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

Testing

There are no formal testing requirements for this assignment. Though, 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:

- **`x = change(x)`:**
  - Similar to with linked lists, do not "morph" a node by directly modifying fields (especially when replacing an intermediary node with a leaf node or vice versa). Existing nodes can be rearranged in the tree, but adding a new value should always be done by creating and inserting a new node, not by modifying an existing one.
  - An important concept introduced in lecture was called `x = change(x)`. This idea is related to the proper design of recursive methods that manipulate the structure of a binary tree. **You should follow this pattern where necessary when modifying your trees**.
- **Avoid redundancy:**
  - If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here.
  - Look out for including additional base or recursive cases when writing recursive code. While multiple calls may be necessary, you should avoid having more cases than you need. Try to see if there are any redundant checks that can be combined!
- **Data Fields:**
  - Properly encapsulate your objects by making data fields in your `ClassificationTree` class private. (Fields in your `ClassificationNode` class should be public following the pattern from class.)
  - Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place.
  - Fields should always be initialized inside a constructor or method, never at declaration.
- **Commenting**
  - Each method should have a comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details

Spec Quiz

Spec Quiz

The following quiz is intended to help make sure you fully understand the assignment specification. We highly encourage you understand the answers to each of the following questions ***prior*** to writing code.

> **NOTE:** Unfortunately, Ed doesn't allow us to reveal quiz answers within an assignment so we've included dropdowns with all the correct answers. However, you should still be attempting each question before checking your answer to verify you fully understand what the spec is asking you to do.

We will not grade any portion of this quiz for completion or correctness. It is simply a resource for you to use in working this assignment should you choose to.

Question 1 *Submitted Jul 22nd 2024 at 1:59:15 pm*

Vocabulary: What word is used to describe any algorithm / process in which computers make probabilistic classifications on inputs.

▼ Expand

This defines a model!

Question 2 *Submitted Jul 22nd 2024 at 1:59:18 pm*

Vocabulary: What is the output of any machine learning classification model?

▼ Expand

This defines a label!

Question 3 *Submitted Jul 22nd 2024 at 1:19:22 pm*

Vocabulary: A split within our dataset stores which of the following

▼ Expand

Splits store both a feature and a threshold! This can be seen within the `Split.java` class

- [ ] Old Input Data
- [ ] Feature
- [ ] Label
- [ ] Threshold

Question 4 *Submitted Jul 22nd 2024 at 1:59:27 pm*

Process: True or false, We should travel *left* if a feature value for our input data is >= the threshold at our current split.

▼ Expand

False, we should travel right if the feature value is >= the threshold

- ( ) True
- ( ) False

Question 5

Process: When "training" a model, we need access to data and known labels for the input data.

▼ Expand

True, "training" a model involves building it from scratch. In order for our newly created model to be accurate, we need to know the correct labels for each datapoint we're learning from.

- ( ) True
- ( ) False

Question 6

Process: Our classification model can only handle datapoints with one feature (e.g `wordPercent` for emails)

Expand

False, there's no limit on how many features our model can handle! The example we work with will only have one, but it's not a requirement.

- ○ True
- ○ False

Question 7 *Submitted Jul 22nd 2024 at 3:47:38 pm*

Files: Which files should you thoroughly read the documentation for in this assignment?

Expand

`Classifiable`, `Classifier`, and `Split` are the only files you need to understand in-depth for this assignment. You're welcome to explore the others but they will not be relevant for the implementation you're writing:

- `Client` provides command-line interaction for a classifier.
- `CsvReader` and `DataLoader` provide some useful methods the `Client` uses to load data from provided files into a format useable by a `Classifier` / `ClassificationTree`
- `Email` is just an example of a `Classifiable` - you should only need to read the `Classifiable` documentation to know the functionality of its methods

- ☐ `Classifiable`
- ☐ `Classifier`
- ☐ `Client`
- ☐ `CsvReader`
- ☐ `DataLoader`
- ☐ `Email`

- [ ] `Split`

Development Guide

Development Guide

Below is a development guide through this assignment, walking you through each process step-by-step and pointing you to relevant methods you should be using in your solution. We recommend following this and making sure that you stop at each of the warning boxes to make sure your solution works correctly before continuing.

We will not grade any portion of this quiz for completion or correctness. It is simply a resource for you to use in working this assignment should you choose to.

Question 1 *Submitted Jul 23rd 2024 at 8:15:25 pm*

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 splits on the dataset and leaf nodes represent classification labels.

Which of the following will be necessary to individually store within your node class?

▼ Expand

Our nodes must store: labels (`String`), splits (`Split`), and old input data (`Classifiable`), as well as left / right pointers since we're making a tree (`ClassificationNode`)!

Splits are necessary for our intermediary nodes. Features and thresholds will be stored within the Split class, so no need to store those individually. Labels are necessary for our leaf nodes when classifying. Old input data is necessary for our leaf nodes when working on the last part of the assignment (the training constructor).

You should store all of these fields in a single node class (don't worry about efficient subclassing / superclassing) even though they won't be used in all cases. This is entirely ok for this assignment.

- [ ] Labels
- [ ] Splits

- [ ] Features
- [ ] Thresholds
- [ ] Old Input Data
- [ ] Models

Question 2

Scanner Constructor

Once you've settled on a node class you're happy with, we recommend working on the Scanner constructor which loads a previously saved file into the tree. 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: wordPercent~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.

> ⚠ 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.

What method should you use to parse the numerical value from a `"Threshold:"` line?

<details>
<summary>Expand</summary>

You should use `Double.parseDouble()` to turn the String "0.125" into a double 0.125
</details>

- ( ) `Integer.parseInt()`
- ( ) `Double.parseDouble()`

- `String.valueOf()`

Question 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 with the format described above.

At this point, test your Scanner constructor and save implementations. You should not move forward in this assignment until these two methods are passing the provided tests.

Looking at the documentation for the `Split` class, what method will be most useful in implementing the `save()` method?

▼ Expand

`toString()` seems to return exactly what we're looking for in terms of printing out our intermediary nodes!

- `getFeature()`
- `getThreshold()`
- `evalute()`
- `toString()`
- `midpoint()`

Question 4

canClassify()

This method should traverse through *all* nodes within the current tree and check if any node contains a feature that doesn't apply to the given Classifiable object. This can be accomplished by checking if the current split's feature is present within the set of all features that pertain to the data. If a single node within the tree contains a feature not present in the data, this method should return false.

⚠ At this point, test your current implementation. You should not move forward in this assignment until the canClassify method is passing.

Looking at the documentation for the `Classifiable` class, what method will be most useful in checking if a split's feature is compatible with a datapoint?

▼ Expand

`getFeatures()` will return a set of all features compatible with a datapoint! From there we can check to see if the feature for this split is within that set. You can then call `Split.getFeature()` and see if it's present in the set to know if things are compatible.

- `get()`
- `getFeatures()`
- `partition()`

Question 5

classify()

Now we can start classifying! This method should traverse through the tree by evaluating splits 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.

⚠ At this point, test your current implementation. You should not move forward in this assignment until the classify method is passing

What method within the `Split` class will help determine if a given `Classifiable` datapoint is above or below the current threshold?

▼ Expand

`evalute()` takes in a `Classifiable` object and returns `true` if it falls below (<) the split and `false` if it falls at or above (>=) the split. This means if this method returns `true`, we should travel *left* and if it returns `false`, we should travel *right.*

- `getFeatures()`

- `evaluate()`
- `toString()`

Question 6

Two List Constructor

Here is where we actually "train" our model, and will likely be the most difficult part of your implementation. Your implementation should follow the following algorithmic approach (copied from the spec - diagrams depicted there):

- Traverse through the current classification tree until you reach a leaf node.
    - If the node's label matches the current label, do nothing (our model is accurate up to this point).
    - If the label is incorrect, create a split between the data used to create the original leaf node* and our current input.
        - Insert a new intermediary node that uses this split to correctly classify the current data and the old data.

Remember to use `x=change(x)` when implementing this method and creating new splits within the dataset.

⚠ At this point, test your current implementation. You should not move forward in this assignment until the two list constructor is passing

Looking at the documentation for the `Classifiable` class, what method will return the best `Split` between the current and provided instances?

▼ Expand

`partition()` does exactly this - it will return the best `Split` for the current and provided `Classifiable` instances

Note that there is no difference in calling `one.partition(two)` and `two.partition(one)`. Both will return `Splits` with the same feature and threshold.

- `get()`
- `getFeatures()`

- [ ] `partition()`

Question 7 *Submitted Jul 22nd 2024 at 3:47:46 pm*

Exceptions

The last step of this assignment is to go through and implement the required exceptions if you haven't already. What's expected is listed below:

- Two list constructor:
  - `IllegalArgumentException` if the provided lists aren't the same size or the lists are empty
- Scanner constructor:
  - `IllegalStateException` if the tree is empty after being loaded from the file
- Classify method:
  - `IllegalArgumentException` if the provided input can't be classified

At this point, test your current implementation. You should not move forward in this assignment until the exception tests are passing

True or false, do you have to implement these exceptions to pass the test cases?

▼ Expand

Can't insert text segments in quizzes on Ed - had to make this a question. The answer is **true**!

- [ ] True
- [ ] False

Reflection

The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

In this reflection, we'll look more closely at large-language models (LLMs) and explore whether or not they're capable of understanding language. ChatGPT is the most well-known example of an LLM. Navigate to chatgpt.com and interact with the model by carrying out some form of conversation with it.

Without the prior knowledge that these responses were machine generated, would you believe them to come from a human being? Why or why not?

> **NOTE:** This is effectively a Turing Test; a famous method of determining whether or not a machine is capable of intelligent behavior.

Question 2

Now, remember that ChatGPT is a machine learning model, meaning it has learned from previously seen examples. Try asking it for the answer to a question it's unlikely to have seen before / memorized (such as a math operation on specific large random numbers). Does it produce the correct response?

If you instead used this interaction to judge whether or not you're interacting with a human, would it change your mind? Why or why not?

Question 3

The next 3 questions will require you reflect on "Section 4: The octopus test" from the paper "Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data". The authors, Bender and Koller attempt a thought experiment to further their argument as to why Machine Learning models (such as ChatGPT) aren't capable of learning language from form (example inputs) alone.

The appropriate section is quoted in the dropdown below for ease of reference, and you're welcome to read the rest of the paper linked above if it interests you.

NOTE: This will likely be a challenging read, do your best and feel free to ask questions on the Ed board if you're confused!

Expand

> In order to illustrate the challenges in attempting to learn meaning from form alone, we propose a concrete scenario. Say that A and B, both fluent speakers of English, are independently stranded on two uninhabited islands. They soon discover that previous visitors to these islands have left behind telegraphs and that they can communicate with each other via an underwater cable. A and B start happily typing messages to each other.
>
> Meanwhile, O, a hyper-intelligent deep-sea octopus who is unable to visit or observe the two islands, discovers a way to tap into the underwater cable and listen in on A and B's conversations. O knows nothing about English initially, but is very good at detecting statistical patterns. Over time, O learns to predict with great accuracy how B will respond to each of A's utterances. O also observes that certain words tend to occur in similar contexts, and perhaps learns to generalize across lexical patterns by hypothesizing that they can be used somewhat interchangeably. Nonetheless, O has never observed these objects, and thus would not be able to pick out the referent of a word when presented with a set of (physical) alternatives.
>
> At some point, O starts feeling lonely. He cuts the underwater cable and inserts himself into the conversation, by pretending to be B and replying to A's messages. Can O successfully pose as B without making A suspicious? This constitutes a weak form of the Turing test (weak because A has no reason to suspect she is talking to a nonhuman); the interesting question is whether O fails it because he has not learned the meaning relation, having seen only the form of A and B's utterances.
>
> The extent to which O can fool A depends on the task — that is, on what A is trying to talk about. A and B have spent a lot of time exchanging trivial notes about their daily lives to make the long island evenings more enjoyable. It seems possible that O would be able to produce new sentences of the kind B used to produce; essentially acting as a chatbot. This is because the utterances in such conversations have a primarily social function, and do not need to be grounded in the particulars of the interlocutors' actual physical situation nor anything else specific about the real world. It is sufficient to produce text that is internally coherent.
>
> Now say that A has invented a new device, say a coconut catapult. She excitedly sends detailed instructions on building a coconut catapult to B, and asks about B's experiences and suggestions for improvements. Even if O had a way of constructing

> the catapult underwater, he does not know what words such as rope and coconut refer to, and thus can't physically reproduce the experiment. He can only resort to earlier observations about how B responded to similarly worded utterances. Perhaps O can recognize utterances about mangos and nails as "similarly worded" because those words appeared in similar contexts as coconut and rope. So O decides to simply say "Cool idea, great job!", because B said that a lot when A talked about ropes and nails. It is absolutely conceivable that A accepts this reply as meaningful — but only because A does all the work in attributing meaning to O's response. It is not because O understood the meaning of A's instructions or even his own reply.

> Finally, A faces an emergency. She is suddenly pursued by an angry bear. She grabs a couple of sticks and frantically asks B to come up with a way to construct a weapon to defend herself. Of course, O has no idea what A "means". Solving a task like this requires the ability to map accurately between words and real-world entities (as well as reasoning and creative thinking). It is at this point that O would fail the Turing test, if A hadn't been eaten by the bear before noticing the deception.

> Having only form available as training data, O did not learn meaning. The language exchanged by A and B is a projection of their communicative intents through the meaning relation into linguistic forms. Without access to a means of hypothesizing and testing the underlying communicative intents, reconstructing them from the forms alone is hopeless, and O's language use will eventually diverge from the language use of an agent who can ground their language in coherent communicative intents.

> The thought experiment also illustrates our point from §3 about listeners' active role in communication. When O sent signals to A pretending to be B, he exploited statistical regularities in the form, i.e. the distribution of linguistic forms he observed. Whatever O learned is a reflection of A and B's communicative intents and the meaning relation. But reproducing this distribution is not sufficient for meaningful communication. O only fooled A into believing he was B because A was such an active listener: Because agents who produce English sentences usually have communicative intents, she assumes that O does too, and thus she builds the conventional meaning English associates with O's utterances. Because she assumes that O is B, she uses that conventional meaning together with her other guesses about B's state of mind and goals to attribute communicative intent. It is not that O's utterances make sense, but rather, that A can make sense of them.

How does the experiment you performed talking with ChatGPT at the beginning of this reflection relate to the thought experiment defined above? What corresponds to A, B, and O?

Question 4

The authors' claim revolves around the idea that understanding the meaning of language relies on "communicative intents". Do you agree that *intent* is necessary for communication? Why or why not?

Question 5

Contrast the perspective above with the word vector example from lecture. Those numerical representations seemingly gained some understanding of how words are related, as evidenced by our comparison examples ("dog is to puppy as cat is to kitten").

With both these examples in mind, do you believe that ChatGPT / other machine learning models are capable of actually understanding language? Why or why not?

Question 6

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

Question 7

What skills did you learn and/or practice with working on this assignment?

Question 8

What did you struggle with most on this assignment?

Question 9

What questions do you still have about the concepts and skills you used in this assignment?

Question 10

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

Question 11

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

Question 12

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

□ Final Submission □

□ Final Submission□

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)