Creative Project 3: Better Than Spam

Specification

New Classifiable

Note that our `Classifier` can work on anything that extends the `Classifiable` class. Let's try it out with some more interesting data! In this extension you'll take an existing dataset, load it into a list of `Classifiable` objects and see how well our model works. Below is a list of datasets we'd recommend messing around with (although you're welcome to explore whatever interests you)

1. Weather (`./data/weather/train.csv`) - predict summary from temperature, humidity, wind speed, etc.
2. Spotify (`./data/songs/train.csv`) - predict genre from danceability, energy, key, etc.
3. Your choice! (It might be worth checking out a website like kaggle. When exploring these datasets, consider whether a machine learning model is actually the best solution to a problem or if it will likely do more harm than good.

Make sure that the dataset you choose has features that are ints/doubles such that they can be classified by our threshold splits! You'll also have to make some changes to the constants in `Client.java` to load the appropriate dataset into whatever `Classifiable` class you write. These changes include implementing an equivalent `toEmail` method that will create an instance of your new object from a row in the `.csv` file. Note in all of these cases that you don't have to use every single possible feature in the `.csv` file, just the one(s) you think will be useful for your model!

> ⚠ We ***HIGHLY*** recommend looking through the provided `Email.java` and making sure you understand all the methods and how they operate before starting this extension.

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:

- All of your fields should be private and each field should not be initialized at declaration.
- Any additional helper methods created, but not specified in the spec, should be declared ***private***.
- 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.

Development Guide

Now's your chance to try running your machine-learning model on some real-world data!

Decide on your Dataset

▼ Expand

Determine what dataset you'd like to use for classification. In doing so you should be thinking about what labels you're trying to predict based on the provided input. We've provided you `weather` and `songs` as examples present in the `data` directory, but you're also welcome to look for datasets elsewhere like kaggle (although it's always worth asking whether or not ML *should* be applied to a dataset before implementing).

Importantly, you'll need to find train / test `.csv` (comma-separated value) files which you can use in your implementation to construct / evaluate your resulting tree. You are also welcome to create a test `.csv` file by reserving values from your training `.csv` file. Please place your chosen dataset in an appropriately named directory within the `data` folder.

Decide on your Features

▼ Expand

Determine which features of the dataset you think would be most beneficial in predicting a corresponding label. There's no right answer here - you can pick one or many features depending on what you're trying to predict. Something important to keep in mind when deciding is that these features must have numerical corresponding values, as splits within our tree store doubles for each feature.

We'd advise avoiding complex feature values (ones that need to be separated by the `SPLITTER` constant like `wordPercent~word`) and instead using simple ones like `windSpeed` / `temperature` / `humidity`.

Keep in mind that this decision can be changed later! However, you should have a good idea of what feature(s) you'd like to include before moving onto the next step. We'd recommend starting with fewer features (1-2) and then adding more should you be unsatisfied with the performance you're seeing in the later steps.

Implement your Classifiable Class

▼ Expand

Now you need to put these decisions into practice! Specifically, you'll be implementing a `Classifiable` class that stores the relevant features for the dataset you've chosen! Remember that implementing this interface requires the following:

A constructor with parameters of your choosing that sets up the initial state.

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

- Returns the numeric value corresponding to the provided feature.

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

- Returns a `Set` containing all features of this datatype

```
public static Classifiable toClassifiable(List<String> row)
```

- Constructs a new instance of this object from a row of the `.csv` file your data is present in represented as a `List<String>`

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

- Returns a Split representing the midpoint between this instance and the provided `other`.
    - This is likely to be the most challenging part of your implementation; however, it is entirely your choice how to determine the midpoint. However, if your implementation has multiple features, this step will likely involve determining a priority you give these features (which is most / least important in determining a midpoint).

It's highly encouraged that you look at `Email` for an example implementation of all of these methods. Note that this implementation uses a complex feature `wordPercent`, meaning its internal state and `partition` algorithm are likely to be more complex than yours!

Test your Model!

▼ Expand

First, you'll have to make a few changes to the provided `Client` class such that it loads and works with your implemented `Classifiable` class.

1. Change the `TRAIN_FILE` and `TEST_FILE` constants to point to the `.csv` files you found in step 1.
    1. Remember, these should be within the `data` directory in an appropriately named subfolder
2. Change the `LABEL_INDEX` constant to the appropriate index of the label you're trying to predict within the training `.csv` file
3. Change the `CONVERTER` constant to reference the `toClassifiable` method you

implemented in step 3

1. This should only involve changing `Email` to the name of the class you implemented!

With these steps completed, you should be able to load and evaluate your new model! Does it perform well / poorly? Why?

At this point you've completed the requirements for this portion of the assignment, but you're welcome to continue iteratively refining your solution. Does adding more features help / harm your result? What about a different subset of features? How about how you're partitioning between two datapoints? All of these things can be changed after the fact - just make sure that you have a working implementation on your final submission!

WARNING: Before submitting, make sure that the `Client` class has been modified such that it loads and tests *your* `Classifiable` implementation (not `Email`). We will run `Client` exactly as provided to determine whether you've met the requirements for this assignment.

New Classifiable

Download Starter Code:

C3_BetterThanSpam.zip

Start by uploading your working implementation to `ClassificationTree` and then work on implementing your `Classifiable` datatype!

Required steps

- Implement your `Classifiable` class following the development guide.
- Make sure your data set is present in the workspace.
- Modify `Client` such that it loads and tests *your* `Classifiable` implementation (not `Email`). We will run `Client` exactly as provided to determine whether you've met the requirements for this assignment.
- Follow the Code Quality Guide and Commenting Guide.

> **NOTE:** The tests provided are surface-level compilation checks. You do not need to pass these tests, but it is still **your responsibility to guarantee that you have a working `Classifiable` and `Client` implementation to receive credit.**

What do I do if I don't have a working `ClassificationTree` solution?

If you do not have a working `ClassificationTree` solution, that's fine! You can use our compiled `.class` files stored in the zip files below and upload those instead of having a `ClassificationTree.java` file. Make sure to delete the `ClassificationTree.java` file so that the `.class` files are actually run!

ClassificationTreeClassFiles.zip