Wentworth Institute of Technology
COMP1050 – Computer Science II
Spring 2015, Derbinsky

Part 4

For this assignment you are to implement a set of class in Java. You must document the classes using Javadoc and turn in the source files, in the appropriate folder structure, as a zip file. You will be evaluated based upon correctness (adhering to the problem description, passing test cases), object-oriented design (e.g. appropriate usage of inheritance, member access control, etc.), and source code documentation (via Javadoc).

Assignment Overview

This assignment is the last building block to your final project and involves implementing classes given an API and a set of unit tests. You are going to implement a hierarchy of classes supporting examples.

0. Assignment Setup

First, download and unzip the `p4-starter.zip` file.

Next, import the contents of the file as a project in Eclipse. You will see red error icons – this is to be expected.

Finally, you should have available for reference the JavaDoc for the whole project (`final-doc.zip`), which includes the classes you must write for this part. (Note: it also includes classes you won't need for this particular project, but that make up the final project.)

1. Your Task

You are required to implement several classes. In doing so, you must adhere to the following general restrictions (additional notes are in subsections below):

- You cannot modify the provided public/protected API in any way.
- You may create variables/methods, but they must be `private`.

The methods you are to implement are all documented, and the test cases provide example usage. The following sections discuss the classes you are to implement.

If you pass all of the test cases, your implementation is in good shape. You may wish to create additional test cases in a separate file, but only turn in the source code for the classes you required to implement.

1.1 `FeatureEntry`

The `FeatureEntry` interface is an abstraction to facilitate a sequence of features, without enforcing an implementation.

1.2 `TestingExample`

The `TestingExample` interface provides the necessary aspects of an example (i.e. enumeration and iteration of features), but does not provide access to the result classification (this is useful for the *testing* phase of classification).

1.3 `Example`

The `Example` base class provides many useful functions, but is still abstract, as it does not require an implementation of feature storage. In case this is confusing, the class itself should support iterating over all features (0, 1, ... $n-1$), but the `toStringIterable` function should support iterating over only non-zero features (these may be the same, but it is up to the implementing subclass to provide an *efficient* solution). Also note that while feature numbers are from 0 to $n-1$, printed features (according to the LibSVM format) are from 1 to n.

1.4 `DenseExample`

Because these are fixed-length examples, it makes sense to back a *dense* example (meaning, mostly non-zero features) with an array. For the `Iterable` interface, you will be required to return an `Iterator` – you should create your own private class that implements the `Iterator` interface and iterates across the array of features. The `Iterator` interface itself requires only two methods: `hasNext` (returns true if there are additional features to iterate over) and `next` (returns an object that implements the `FeatureEntry` interface you defined above). I would recommend that your custom `Iterator` itself implements the `FeatureEntry` interface, and so you get pieces of functionality in a single class.

1.4 `SparseExample`

In many datasets, it is common that there are many features, but most examples make use of relatively few – this is the ideal circumstance for the `SparseExample` class, which stores a sorted pairing of feature number-value pairs. Exploiting this representation can be a big memory and time savings for large datasets.

You will need to devise a strategy for storing the non-zero feature/entry pairs. I would recommend looking into the `TreeMap` class[1], which stores its keys in sorted order (perfect for iterating across features in order!). You will also need to implement two different kinds of iterators for the sparse example: one that goes across *all* features, and one that just hits the non-zero features. Neither are complex in terms of code if you think them through ahead of time.

2. Opportunities

The Euclidean-based distance function you wrote in Part 3 is quite wasteful/slow in datasets that are full of sparse examples (i.e. there may be lots of accesses/multiplications of zeroes). For extra credit, submit a version of that class that works in all cases, but, if both arguments are actually `SparseElement` objects (see the `instanceof`[2] operator), operates much more quickly.

[1] See `http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html`
[2] `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html`