Lab - Find Author

Adapted from the nifty assignment “Authorship Detection” by Michelle Craig, University of Toronto. (Presented at SIGCSE 2013, Denver Colorado)

Version 012315

Throughout this lab, you must use the highest level of abstraction possible. Instance fields, for example, should be of an interface type whenever possible, viz. use `List` instead of `ArrayList.` You *MAY NOT* use the primitive type `char` or the class `Character.` Unless otherwise explicitly stated, you may use only library classes or methods that are listed in the AP Quick Reference guide. Permission to locally use a method or class that is not listed in the AP Quick Reference guide does not mean that you are free to use the method or class elsewhere in your code.

Poorly designed and poorly documented code *at any stage of the project* will have a proportional adverse effect on your grade. Note that your teacher may look at your code at *any* time during the lab and assess your progress at that time.

Introduction

Automated authorship detection is the process of using a computer program to analyze a large collection of texts, one of which has an unknown author, and making guesses about the author of that unattributed text. The basic idea is to use different statistics from the text -- called "features" in the machine-learning community -- to form a linguistic "signature" for each text. One example of a simple feature is the number of words per sentence. Some authors may prefer short sentences while others tend to write sentences that go on and on with lots and lots of words and not very concisely, just like this one. Once we have calculated the signatures of two different texts we can determine their similarity and calculate a likelihood that they were written by the same person.

Automated authorship detection has uses in plagiarism detection, email-filtering, social-science research and as forensic evidence in court cases. Also called authorship attribution, this is a current research field and the state-of-the-art linguistic features are considerably more complicated than the five simple features that we will use for our program. But even with our very basic features and simplifications, your program may still be able to make some reasonable predictions about authorship.

Some Definitions: What really is a sentence?

Before we go further, it will be helpful to agree on what we will call a sentence, a word and a phrase. Let's define a **token** as follows:

- A 'word' which is defined as a non-empty sequence of characters that begins with an alpha character and then consists of alpha characters, numbers, the single quote character "'", or the hyphen character "-". Words are converted to lower case.
- An 'end of sentence' delimiter defined as any one of the characters ".", "?", "!".
- An end-of-file token which is returned when the scanner is asked for a token and the input is at the end-of-file.
- A phrase separator which consists of one of the characters "," (comma),":" (colon) or ";" (semicolon).
- A digit.
- Any other character not defined above.

For the purposes of this assignment, we will consider a **sentence** to be a sequence of characters that (1) is terminated by (but doesn't include) the characters ! ? . or the end of the file, (2) excludes whitespace on either end, and (3) is not empty. Consider the following:

```
this is the\nfirst sentence. Isn't\nit? Yes ! !! This \n\nlast bit :) is also
a sentence, but \nwithout a terminator other than the end of the file\n
```

By our definition, there are four "sentences":

Sentence 1 "this is the\nfirst sentence"
Sentence 2 "Isn't\nit"
Sentence 3 "Yes"
Sentence 4 "This \n\nlast bit :) is also a sentence, but \nwithout a terminator other than the end of the file"

Notice that:
The sentences do not include their terminator character.
The last sentence was not terminated by a character; it finishes with the end of the file.
Sentences can span multiple lines of the file.

Phrases are defined as non-empty sections of sentences that are separated by colons, commas, or semi-colons. The sentence prior to this one has three phrases by our definition. This sentence right here only has one (because we don't separate phrases based on parentheses).

We realize that these are not the correct definitions for sentences, words or phrases but using them will make the assignment easier. More importantly, it will make your results match what we are expecting when we test your code. You may not "improve" these definitions **or your assignment will be marked as incorrect.**

Linguistic features we will calculate

The first linguistic feature recorded in the signature is the **average word length**. This is simply the average number of characters per word, calculated after the punctuation has been stripped. Notice that the comma and the final period are stripped but any hyphen characters or underscore characters are counted.

Type-Token Ratio is the number of different words used in a text divided by the total number of words. It's a measure of how repetitive the vocabulary is. Note that "this","This","this," and "(this" are *not* counted as different words.

Hapax Legomana Ratio is similar to Type-Token Ratio in that it is a ratio using the total number of words as the denominator. The numerator for the Hapax Legomana Ratio is the number of words occurring exactly once in the text.

The fourth linguistic feature your code will calculate is the **average number of words per sentence**.

The final linguistic feature is a measure of **sentence complexity** and is the average number of phrases per sentence. We will find the phrases by taking each sentence, as defined above, and splitting it on any of colon, semi-colon or comma.

Exercise – Scanning and tokenizing the input

Create a new project called `FindAuthor.` From the supplied starting code, add the `Scanner` class.

The `Scanner` uses a one character look-ahead scheme in which the next character in the input stream is fetched in advance of beginning the processing of the character. This strategy allows for easy error detection when the next character can be predicted. The method `getNextChar()`accomplishes this look-ahead and it is called *ONLY* by the constructor and the `eat()` method.

A constructor and the `getNextChar()`methods are written for you.

In the `Scanner` class, add a private method called `eat` which takes in a `String` object and then compares it to the `currentChar` instance field. If they match, the method advances the input stream by calling `getNextChar().` Otherwise, it throws an `IllegalArgumentException` with information that would allow a user to begin debugging. Note that the `eat` method (and the constructor) are the ONLY methods that call `getNextChar()`. DO NOT CALL `getNextChar()` ANYWHERE ELSE.

We will use a number of private helper methods to help with the task of tokenizing the input stream. Each of these methods takes in a `String` object that is expected to contain a single character. Each of these methods must be written using a single line of code that begins with the keyword `return` and returns a `boolean` value.

Add and implement the following methods in the `Scanner` class:

1. A private method that returns true if the `String` parameter is a letter.
2. A private method that returns true if the `String` parameter is a digit.
3. A private method that returns true if the `String` parameter is special character consisting of a single quote or a hyphen.
4. A private method that returns true if the `String` parameter is a phrase terminator.
5. A private method that returns true if the `String` parameter is a sentence terminator.
6. A private method that returns true if the `String` parameter is white space.

Next add a public method called `hasNextToken()` which returns true if there are more tokens in the input stream. There are more tokens if the input stream is not at end-of-file.

Add an immutable class called `Token` to your project. The `Token` class constructor takes in a `Scanner.TOKEN_TYPE` object and a value which is a `String` object. Write the constructor.

Next, add an appropriate accessor method to the `Token` class.

Add a `toString()` method that overrides the `toString()` method in the `Object` class. In the body of the `toString()`method, add appropriate code that will return a meaningful `String` representation of the current `Token` object.

Override the `equals()` method from the `Object` class within your `Token` class. The `equals()` method will allow `Token` objects to be compared for equality. Carefully define what it means when one `Token` equals another `Token`.

Optionally override the `hashcode()` method from the `Object` class.

Write the `Scanner` class `nextToken().` Make certain that each `Token` of type `Scanner.TOKEN_TYPE.WORD` is returned lower case. You may use the String method `toLowerCase()` to accomplish this.

Recall that a token is defined as:

- A 'word' which is defined as a non-empty sequence of characters that begins with an alpha character and then consists of alpha characters, numbers, the single quote character "'", or the hyphen character "-".
- An 'end of sentence' delimiter defined as any one of the characters ".", "?", "!".
- An end-of-file token which is returned when the scanner is asked for a token and the input is at the end-of-file.
- A phrase separator which consists of one of the characters ",",":" or ";".
- A digit.
- Any other character not defined above.

Test the `Scanner` class completely. To help you with your testing, you may use the `StringReader` class. For example:

```
StringReader reader = new StringReader("This is a string");
Scanner scanner = new Scanner(reader);
```

Of course, the method `main` which accomplishes this testing is never located within the `Scanner` class. Instead, it is in its own class named according to the style guide and properly documented.

Have your `Scanner` and associated tester checked off by your teacher **before** proceeding.

Exercise – Document, Sentence and Phrase

When computing document statistics, all punctuation is discarded. We are interested only in the sequences of words that form a phrase and the sequences of phrases that form a sentence.

Create a class called `Phrase.` A `Phrase` consists of a group of `Token` objects, each containing a single word. You will need a data structure to store individual `Token` objects, and you will need to efficiently traverse the data structure as well as quickly access individual elements. You ***must*** justify your choice within the class documentation. Create a constructor that initializes your private data.

Add the following methods to your `Phrase` class:

- A method that allows you to add `Token` objects to the internal data structure. Be sure to document the Big O performance.
- A method that returns a copy of the internal data structure holding the sequence of `Token` objects. A shallow copy is not acceptable.
- A `toString` method that provides a `String` representation of this `Phrase.`

DO NOT ADD ANY OTHER METHODS.

Create a class called `Sentence.` A `Sentence` consists of a sequence of `Phrase` objects. You will need a data structure to store individual `Phrase` objects, and you will need to efficiently traverse the data structure as well as access individual elements. Choose appropriately and justify your choice, including Big O performance. Create a constructor which initializes the private data within the class.

Add the following methods to your `Sentence` class:

- A method that allows you to add `Phrase` objects to the internal data structure.
- A method that returns a copy of the internal data structure holding the sequence of `Phrase` objects. A shallow copy, that is returning a reference to the private data, is not acceptable.
- A `toString` method that provides a `String` representation of this `Sentence.`

DO NOT ADD ANY OTHER METHODS.

Create a class called `Document.` This class will be responsible for directing the scanning and parsing of a document. It will encapsulate all of the sentences found in the document. The document constructor should take in a `Scanner` object. It should also initialize a data structure for `Sentence` objects. You will need a data structure to store individual `Sentence` objects, and you will need to efficiently traverse the data structure as well as access individual elements. Choose appropriately and justify your choice of data structure within the class comments. The data structure must be initialized in the constructor. You will also need an instance field to store the current value of the `Token` from the scanner.

We will use a one `Token` look-ahead scheme similar to the `Scanner` one character look-ahead. Write the method `getNextToken` which will request a `Token` object from the `Scanner`. Write the method `eat` which takes in a `Token` object and compares it to the current `Token`. If they match then proceed to get the next token from the input. Otherwise, throw a

`RuntimeException`. Be sure to provide debug information (a `String` such as 'illegal token' without any other information is useless). The `eat` method and the constructor are the *ONLY* methods that call `getNextToken`. Be sure your code adheres to this important restriction.

Add the following methods to the `Document` class:

- A method called `parsePhrase` which takes in no input parameters and accepts `Token` objects of type `WORD` and adds them to the current `Phrase.` It uses `currentToken`. It stops parsing the phrase when the `Token object is END_OF_PHRASE, END_OF_FILE,` or `END_OF_SENTENCE.` Other `Token` types are discarded.
- A method called `parseSentence` which has no input parameters. The method `parseSentence` parses individual phrases from the input stream until the end of a sentence is reached or the end of file is reached. It uses `parsePhrase` to accomplish its work and it returns a `Sentence` object.
- Add a method called `parseDocument` which will continue to parse sentences from the input stream until end of file is reached. The method `parseDocument` will skip any leading tokens that are not of type `WORD.`
- An accessor method for the data structure that stores the individual `sentence` objects. A shallow copy is acceptable.

Notice that the parsing technique descends from `Document` to `Sentence` to `Phrase.` This is a simplified version of a parsing technique called 'recursive descent'. Our parser would become recursive if, for example, a `Phrase` object could contain a `Sentence`.

DO NOT ADD ANY OTHER METHODS.

Exercise – Test your code

You have now completed the code for parsing an input text and dividing it up into words, phrases and sentences. You now need to fully test your code. Design and then code a tester that completely tests your code and then demonstrate it to your teacher. Be prepared to justify your stated test strategy and methodology.