Text Generator

Due Sunday, September 17, 2017

Objectives

For this assignment, you will:

- Gain practice using Java generics
- Gain practice using the `ArrayList` and `Association` classes
- Gain proficiency in using objects to build more complex data structures

Description

In this assignment, we will use a basic technique in Artificial Intelligence to automatically generate text. You will write a program that reads in a piece of text and then uses that text as a basis to generate new text. The method for generating new text uses simple probability.

First, we read in a piece of text word by word (we consider punctuation symbols to be words) keeping track of how often each three-word sequence (trigram) appears. For example, consider the following excerpt from Rudyard Kipling's poem "If":

> If you can keep your head when all about you
> Are losing theirs and blaming it on you,
> If you can trust yourself when all men doubt you,
> But make allowance for their doubting too;
> If you can wait and not be tired by waiting

In this excerpt, the trigrams are: "if you can", "you can keep", "can keep your", "keep your head", "your head when", "head when all", etc.

Once we have counted all of the trigrams, we can compute the probability that a word w_3 will immediately follow two other words ($w_1 w_2$) using the following equation:

$$p(w_3|w_1, w_2) = \frac{n_{123}}{n_{12}}$$

where n_{123} is the number of times we observed the sequence $w_1 w_2 w_3$ and n_{12} is the number of times we observed the sequence $w_1 w_2$.

For example, let's compute the probability that the word "can" will immediately follow the words "if you". In the excerpt above, the sequence "if you can" occurs three times. The sequence "if you" also occurs three times. So the probability is given by,

$$p(can|if\ you) = 3/3 = 1$$

The probability that any other word comes immediately after "if you" is 0. Consider another example. In the excerpt above, the words "you can" appear 3 times. The first time followed by "keep", the second time by "trust" and the last time by "wait". Thus,

$$p(keep|you\ can) = 1/3$$
$$p(trust|you\ can) = 1/3$$
$$p(wait|you\ can) = 1/3$$

Again, the probability that any other word besides "keep", "trust", or "wait" appears after "you can" is zero.

Once we have the text processed, and stored in a data structure that allows us to compute probabilities, we can generate new text. We first pick two words to start with (for example, the first two words in the input text). Given these two words, and the probabilities computed above, we use a random number generator to choose the next word. Continue this process until the new text is 400 words long (including punctuation). When printing the new text, please generate a new line after every 20 words so that you don't just generate one very long, unreadable, line.

Note: There may be two words that were seen in the text but were never followed by any word. For example, in the excerpt of the Kipling poem above, the words "by waiting" occur once but are never followed by a word (since they are the last two words in the excerpt). In this case, you can either stop generating words or feel free to think up a more creative solution.

Warning: This assignment is significantly harder than the last one and will take a lot of time. Please start early to save yourself lots of grief!

Classes

`FreqList` class

We suggest that you write the `FreqList` class first. `FreqList` should contain an array list of `Associations`. `ArrayList` is part of the `java.util` package with documentation found in the Java 8 Javadoc pages linked to from the handout page on the course website. The `Association` class is part of Bailey's `structure5` package. Javadoc for this class is also available from a different link on the handout page.

An `Association` has both a key and a value. The key is a word and the value is the number of times the word occurs. When a word is added, if it already occurs in the array list then its value (i.e. its frequency) is incremented by 1. If it doesn't exist, add the word to the array list with a value (i.e. frequency) of 1.

Your `FreqList` class should have an instance variable that keep tracks of the number of words added. This is equivalent to the sum of all the values (i.e. frequencies) of all the `Associations` in the array list.

The `FreqList` class should also have a method with the following definition:

```
public String get(double p)
```

This method takes a double p as an input and returns a word from the array list. The input p must be between 0.0 and 1.0, otherwise the method should throw an `IllegalArgumentException`. How can we use p to generate a word? In our example above, the `FreqList` for the words "you can" would look like: {<"keep", 1>, <"trust", 1>, <"wait, 1>}. The sum of all of the frequencies is 3. Thus, we will return "keep" whenever $0 \leq p < 1/3$, "trust" whenever $1/3 \leq p < 2/3$ and "wait" whenever $2/3 \leq p \leq 1$. If the frequency list is empty, this method should return an empty string.

Write this class and test it thoroughly by adding a main method to make sure all methods work correctly. We suggest printing out a representation of the frequency list first to make sure that the table is correct before attempting to write or test the probabilistic `get` method.

`TextGenerator` class

The `TextGenerator` class should also contain an array list of `Associations`. Again, an `Association` has a key and a value. In this case, the key is a sequence of two words (e.g., "you can"). The value is an object of type `FreqList`. We have included a class `StringPair` that can be used to represent a sequence of two words. Thus, the array list in `TextGenerator` should have interface type

`List<Association<StringPair, FreqList>>}.`

We have provided you with startup code in the main method of class `TextGenerator` that will pop up a dialog box to allow the user to choose a file containing the input text. We have also provided the headers of two methods for that class.

The first method, `enter(s1, s2, s3)`, will be used to build the table. The three `String` parameters are used to build the table (list) that will be used later to generate random text.

Suppose the table you are building is named `table` and the input starts with "This is the time for". Then you will call `table.enter("This", "is", "the")`, which should update the entry for the word pair of "This" and "is" to record that "the" is a possible word to follow the pair. More carefully, if there is no entry for that word pair, then it will create one for that pair with an empty frequency list, and then add "the" occurring once to that frequency list. If the word pair is already there, update the frequency list to record the added occurrence of "the". Having done this, continue with `table.enter("is","the","time")`, then `table.enter("the","time","for")`, and continuing in the same way if there is more text.

We have included some text files (ending with suffix "txt") in the assignment folder that you can use to test your program. We've tried to pick files with sufficient repetition of triples.

After the input has been processed to build the table, you should generate new text. You may start with a fixed pair of words or choose two words randomly. The method `getNextWord(s1,s2)` uses the table generated from the input to return a randomly chosen word from the frequency list associated with the word pair of s1 and s2. (Of course it should choose the word using the probabilities as discussed earlier.)

Generate and print a string of at least 400 words so that we can see how your program works. When printing the text, please generate a new line after every 20 words so that you don't just generate one very long, unreadable, line.

Finally, we'd like you to print the table of frequencies in some reasonable fashion so we can check to see if your table is correct on our test input. Here are some lines of output based on the lyrics for Dylan's "Blowin' in the wind":

```
The table size is 122
<Association: <how,many>=Frequency List: <Association: roads=1><Association: seas=1>
    <Association: times=3><Association: years=2><Association: ears=1><Association: deaths=1>>
<Association: <many,roads>=Frequency List: <Association: must=1>>
<Association: <roads,must>=Frequency List: <Association: a=1>>
<Association: <must,a>=Frequency List: <Association: man=2><Association: white=1>>
...
```

Note that we cheated and manually wrapped the first line of output so that it is readable. Your output need not wrap. Note that most of this output comes from the ***`toString`*** *methods of* ***`Association`*** *and* ***`ArrayList`****, though we did some extra work to print each entry in the array list on a separate line.*

This output indicates that after the word pair, "how many", the words "roads", "seas", "times", "years", "ears", and "deaths" each occurred once. Whereas after "many roads", only "must" occurred and it only occurred once.

Warning: As a general rule it is good to be as specific as possible with import statements. Thus, when you import the package with the class Random, use

```
import java.util.Random;
import java.util.ArrayList;
import structure5.Association;
```

If your program included "`import java.util.*;`" along with "`import structure5.*` the program might get confused as to which version of classes it should use as there are several classes in java.util that have the same names as those in structure5. It is best to be safe even when we don't have conflicts.

`WordStream` class

This class is already implemented for you. It processes a text file and breaks up the input into separate words that can be requested using the method `nextToken`. In the `main` method of the `TextGenerator` class, there is an object of type `WordStream` called `ws`. You can get successive words from `ws` by calling `ws.nextToken()` repeatedly. Before getting a new word, always check that there is a word available by evaluating `ws.hasMoreTokens()`, which will return true if there are more words available. If you call `nextToken()` when there are no more words available (i.e., you've exhausted the input), then it will throw an `IndexOutOfBoundsException`.

`StringPair` class

This class is already implemented for you. This class provides objects that consist of a pair of strings. These should be used as the keys to your `Association`s.

`Pair` class

This class is already implemented for you. This is a generic class that `StringPair` extends by specializing the type variables to both be `Strings`. You should not directly use this class in your program. Instead use the more compact name `StringPair` from the more specialized class defined above. We provided this class only to show how easy it is to define a generic class that can be instantiated in many ways.

`Association` class

This class is part of Bailey's `structure5` package. There is one piece of information about the `Association` class that we did not highlight in class that you will likely find very useful. The `equals` method in Association only compares key values. That is if two objects from class `Association` are "equal" according to the method, then their keys are the same, but their values may be different. Thus, to find out if an association with a given key is in an arraylist (and where it is), just create a new association with that key and use the `indexOf` method to determine its index in the list. If -1 is returned, then it is not there, whereas if a non-negative integer is returned then you get the index of an association with that key in the arraylist.

Getting Started

1. Think carefully about the design of this program *before* sitting down at a computer. What other methods and instance variables might the `FreqList` class need? For the `get` method in the `FreqList`

class, take a piece of paper and create different examples of frequency lists. For each example, work out what the `get` method would return for different values of p (e.g. if the frequency list contained 4 words that each occurred once and $p = 0.45$, what word would be returned?)

2. Using Finder (or scp if youre working remotely), copy the starter directory `/common/cs/cs062/assignments/assignmen` directly into your Eclipse workspace.

3. Next in Eclipse, select File → Import and target the folder you just copied.

4. Finally, add the BAILEY variable to the projects Java Build Path (go to Project → Properties with the project selected and open the Java Build Path tab).

5. Refresh the project in Eclipse.

6. You are now ready to get started! We recommend starting with the `FreqList` class. Again, *try and develop incrementally.* That is, get one small piece working and then move on to another piece. Add a `main` method to the `FreqList` class in order to test and ensure the correctness of your code.

Grading

You will be graded based on the following criteria:

criterion	points
FreqList	4
TextGenerator	3
Prints FreqList	2
Prints at least 400 words of generated text	2
Appropriate comments (including JavaDoc)	2
Appropriate use of data structures	2
Style and formatting	2
General correctness	2
submitted correctly	1
extra credit	1

NOTE: Code that does not compile will not be accepted! Make sure that your code compiles before submitting it.

Submitting Your Work

1. Test your classes thoroughly before turning in your project. Feel free to use the text files in the assignment folder to test your program. While you may not compare code with other students, you may compare the contents of the tables you generate.

2. Make sure you have added appropriate Javadoc comments to your code.

3. Export your project from Eclipse and submit it using the instructions at `http://www.cs.pomona.edu/classes/cs062/handouts/eclipse_submission.pdf`. Please follow these very carefully. If you do not, your program will NOT be submitted correctly and you will not receive credit for your assignment (and you and we will all be very sad!). Don't forget to fill out the `asg02.json` file! In particular, don't forget to set the "ec" field to true if you did extra credit.

Extra Credit Opportunities

There are many ways in which you might extend such a program. For example, as described, word pairs which never appeared in the input will never appear in the output. Is there a way you could introduce a bit of "mutation" to allow new pairs to appear?

These "trigrams" of words are used in natural language processing in order to categorize kinds of articles and their authors. Think about how you might use tables like those given here to help determine if two articles are written by the same author.

In case you do attempt an extra credit extension, don't forget to set the "ec" field to true in the `asg02.json` file so that we know you did it. You should also indicate in comments wherever you attempted extra credit.