Homework 9: Machine Learning, Stage I

Consider the following romantic lines:

> "This is the sun! Facing him, rowing lustily.
>
> That goes down the drama, is practising "How now, who calls?"
>
> She takes his hand. He goes to her in agony. He finds a seat, and
>
> The audience is quiet and her spirit stronger than the sea."

Now consider the following scholarly text:

> "Yet every breeder would, according or not, as far as instincts are less vigorous and authority, not to hesitating freely enter into the gizzard. Looking to oscillations of correlated to that animals exhibits an affinity from any regions. And the seeds of the cells of the other forms will bring natural selection. In our diagram each letter is direct action. But what an entire and absolutely perfectly obliteration of the skulls of young mammals was carefully developed on fossils."

Can you tell who wrote these texts? If you guessed some moron relatives of Shakespeare and Darwin, then, well, you are almost correct.

Read the texts again. You will notice that, on the one hand, they sound like English. They even have distinct and very different styles. On the other hand, the contents of these texts are sheer nonsense (although we dare say that some journals will happily publish them).

As you may have guessed, these texts were generated by a computer program. In the first example, the program read a real 120-page Shakespeare play, and taught itself to write "like" Shakespeare. In the second example, the program read the 700-page classic *The Origin of Species*, and taught itself to write "like" Darwin. In this project you will write such a text generation program. You will then be able to train it to write books by any one of your favorite authors, as long as some of their works are freely available from such websites as Project Gutenberg.

The program that you will write is based on a machine learning algorithm. Using statistical techniques known as Markov Processes and Monte Carlo methods, the program will first "learn" the style of any text that you feed it (assuming that the text will be sufficiently long). The program will then go on to happily generate nonsense texts written in the same style, and in any desirable length.

As it turns out, this program is much more than a cute gimmick. The principles that underlie this program are heavily used today in numerous applications, ranging from spell checkers to language translators to the technique that Google uses to complete your search phrases. Some researchers even use these algorithms to train computer programs to write computer programs.

We will approach the development of this machine learning program in two stages. In Stage I, you will write several methods that operate on strings and linked lists. In Stage II, you will use these tools to complete the rest of the project.

Background

We wish to count how many times each character appears in a given string, and represent the resulting counts in a linked list. For example, consider the string "committee_", where '_' stands for the space character. The character counts in this string can be represented by the following linked list:

((‘c’,1, 0, 0) (‘o’, 1, 0, 0) (‘m’, 2, 0, 0) (‘i’, 1, 0, 0) (‘t’, 2, 0, 0) (‘e’, 2, 0, 0) (‘_’, 1, 0, 0))

Each node in the list is an object that has four fields: a character (`chr`), the number of times that the character appears in the string (`count`), and two more fields (`p` and `cp`) which are presently set to 0. We'll return to these fields later in this document. Till then, ignore them.

I-0. Getting started

Our main data structure in this project is a linked list of `CharProb` objects. Said otherwise, each list element is an instance of the `CharProb` class. Review the code and documentation of the supplied `CharProb` class, ignoring the opening comment about computing `p` and `cp`. Note that all the fields in `CharProb` are package-private.

I-1. Searching lists

Review the `indexOf` method and the code of the supplied `test1` method. Note that the code makes use of the `add` method of Java's `LinkedList` class. If you are not sure what this method is doing, consult Java's `LinkedList` class API. Implement the `indexOf` method and run the code of `test1`. Implementation tip: one way to iterate (process all the elements of) a list represented by a `LinkedList` object is to use the `size` and `get(int)` methods exposed by the `LinkedList` class.

I-2. Displaying lists

In this project we use Java's built-in `LinkedList` class. Therefore, we have no control over its `toString` implementation. In fact, an inspection of the `LinkedList` API reveals that this class doen't have a `toString` method. This may seem surprising, but actually it makes perfect sense, as you'll see when we learn inheritance later in the course.

To make a long story short, we will write a `toString` method of our own. Review the `toString` method and the code of the supplied `test2` method. The `toString` method takes a linked list as a parameter, and uses a loop to iterate the list and print all its elements (objects), one by one. Implementation tip: inspect the `toString` method of the `CharProb` class, and note that each `CharProb` object knows how to print itself. Therefore, our `toString` method can be implemented quite simply: iterate the list, and tell each element to print itself.

Note that we are using an interesting design setting: the linked list, which is an object, is printed "from the outside" using a static `toString` method (function). At the same time, each list element, which is also an object, is printed using a typical object-oriented `toString` method. We say that the ability to print is *encapsulated* in the `CharProb` class. *Encapsulation* is a very important

object-oriented design objective. In general, one should strive to implement as much functionality as possible as *member methods* in the class from which the objects are derived. This way, the objects will know how to take care of themselves. You will understand this principle better once you complete the implementation of the `toString` method.

I-3. Building lists

Review the `buildList(String)` method and the code of the supplied `test3` method. Then implement and test the `buildList(String)` method. This method creates an empty linked list, and then goes on to populate it with `CharProb` objects, one for each unique character that appears in the string. For example, if the string is "committee_", the method will produce the list documented in the `test3` method. Implementation tips: For each character in the string, check if the character exists in the list. If so, increment its `count` field. Otherwise, construct a new `CharProb` object and adds it to the list. You will find the `indexOf` method quite handy here.

The Plot Thickens

Recall that our main data structure is a linked list of 0 or more `CharProb` objects. Each one of these objects has four properties (also known as field values) named `chr`, `count`, `p`, and `cp`. The `count` field contains the number of times that `chr` appears in a given string, and the `p` and `cp` fields contain (for now) zeros. For example, the string "committee_" is associated with the following list:

((‘c’,1, 0, 0) (‘o’, 1, 0, 0) (‘m’, 2, 0, 0) (‘i’, 1, 0, 0) (‘t’, 2, 0, 0) (‘e’, 2, 0, 0) (‘_’, 1, 0, 0))

An inspection of the list reveals that the input string contains a total of 10 characters. We can now use this parameter, as well as the list itself, to compute, and set, the probabilities `p` and `cp` of every list element. Following this calculation, the list becomes:

((‘c’,1, 0.1, 0.1) (‘o’, 1, 0.1, 0.2) (‘m’, 2, 0.2, 0.4) (‘i’, 1, 0.1, 0.5) (‘t’, 2, 0.2, 0.7) (‘e’, 2, 0.2, 0.9) (‘_’, 1, 0.1, 1.0))

Technically speaking, this list specifies what is known as a *probability distribution*. A probability distribution describes a set of exhaustive and mutually exclusive events, along with the probability of each event. In our case, the list can be viewed as specifying the probabilities of drawing different characters at random from the given string. For example, the probability of drawing the character ‘c’ is 0.1, and so is the cumulative probability of this event. The probability of drawing an ‘o’ is 0.2, and the cumulative probability of drawing either ‘c’ or ‘o’ is 0.2. The probability of drawing ‘m’ is 0.2, and the cumulative probability of drawing ‘c’, or ‘o’, or ‘m’ is 0.4. And so on. The last entry in the list says that, in this particular string example, the probability of drawing the space character ‘_’ is 0.1, and the cumulative probability of drawing either ‘c’, or ‘o’, or ‘m’, or ‘i’, or ‘t’, or ‘e’, or ‘_’ is 1.0.

Let i be the element number in a probabilities list containing n elements. Note that $cp_0 = p_0$, and that for each $i > 0$ we have $cp_i = cp_{i-1} + p_i$. If you run this calculation correctly for all the list elements, you are guaranteed to get $cp_{n-1} = 1$.

I-4. Calculating probabilities

Preview the `calculateProbabilities` method and the supplied `test4` method. The `calculateProbabilities` method receives a linked list as a parameter, and computes and sets the

`p` and `cp` fields of all the list elements, each being a `CharProb` object. Note that the first thing that you have to do is iterate the list and compute how many characters exist in total. You can then use this factor to compute, and set, the values of `p` and `cp` of all the list elements. Implementation tip: since all the fields of `CharProb` are package-private, you can access and set them directly.

I-5. Getting a random character

Why are we going through all this trouble? Well, we are setting the stage for the machine learning algorithm that will be introduced in Part II of this project. One element of this algorithm calls for drawing characters at random from a given string, according to the number of times that the characters appear in that string. For example, when drawing at random characters from a typical English text (like this paragraph), the character '`e`' should be drawn more frequently that the character '`q`'.

The probability list that we've been building all along is perfectly suited to support this operation. Note that the list has a length, which, by construction, is also the number of unique characters that appears in the given string. For example, the length of the list associated with the string "`committe_`" is 7, and the indexes of the seven unique characters in the list are 0, 1, 2, …, 6. Our goal is to draw a character from this list at random, according to the character's probability. For example, if we repeat this experiment many times, the number of times that we draw '`m`' should be about twice as much as the number of times that we draw '`c`'.

So how can we generate a probabilistic event according to its probability? This can be done using a technique known as *Monte Carlo*. We start by drawing a random number in [0,1). Let's call the resulting random number *r*. We then iterate the list, reading the cumulative probabilities as we go along. We stop at the element whose cumulative probability is greater than *r*, and return the character of this element. For example, *r* = 0.38, we return '`m`'; if *r* = 0.55, we return '`t`', and so on. Note that the maximal *r* that we can possibly get is 0.99999…, in which case we return (in this particular case) the space character '`_`'. If all this sounds vaguely familiar, it's because we used the same Monte Carlo technique when we implemented the pageRank algorithm (Lecture 5-1, slides 19-21).

Implement the `getRandomChar` method, and run the supplied `test5`. Note that this test is naive, since it simply draws one random character from the list. The `getRandomChar` method requires stress-testing, and the only way to do it is to repeat this experiment many times, and observe the results. Design, document, and implement an experiment that carries out such a stress test, and put its code in the supplied `test5` method.

Submission: There is no need to submit anything for now. You will submit the entire project after we publish Part II.

Deadline: For your own sake and sanity, you must complete this part of the project no later than January 9, 2017.