Markov

Read the sections in their entirety before you start programming!

Introduction

The mathematical root of this project is from a 1948 monolog by Claude Shannon, A Mathematical Theory of Communication which discusses in detail the mathematics and intuition behind this assignment. This article was popularized by AK Dewdney in both Scientific American and later reprinted in his books that collected the articles he wrote.

In 2005 two students at MIT had a randomly generated paper accepted at a conference. Their paper-generating tool, SCIgen, has a Wikipedia entry with links to the program. In 1984 an Internet Personality named Mark V Shaney participated in the then chat-room-of-the time using randomly generated text as described in this Wikipedia entry.

In December 2008 Herbert Schlangemann had an auto-generated paper accepted and published in an IEEE "conference" (quotes used judiciously).

Overview

You'll do two things for this assignment.

1. Improve the performance of code that generates random text based on predicting characters. For this you'll use a `Map` to store information rather than (re)computing the information.

2. Write a new random-text generation program based on words rather than characters.

You'll be provided with code that uses a brute force approach to generate random text using an order-k Markov model based on characters. You'll first improve the code to make it more efficient, then you'll write a new model based on words rather than on characters.

The term *brute force* refers to the characteristic that *the entire text that forms the basis of the Markov Model is rescanned to generate each letter of the random text.*

For the first part of the assignment you'll use a map data structure so that the text is scanned only once. When you scan once, your code will store information so that generating random text requires looking up information rather than rescanning the text.

Background on Markov Models

An order-*k* Markov model uses strings of *k*-letters to predict text, these are called *k-grams*. An order-2 Markov model uses two-character strings or *bigrams* to calculate probabilities in generating random letters. For example suppose that in the text we're using for generating random letters using an order-2 Markov model the bigram "**th**" is followed 50 times by the letter 'e', 20 times by the letter 'a', and 30 times by the letter 'o', because the sequences "the", "tha" and

"tho" occur 50, 20, and 30 times, respectively while there are no other occurrences of "th" in the text we're modeling.

Now suppose that in generating random text we generate the bigram "th" and based on this we must generate the next random character using the order-2 model. The next letter will be an 'e' with a probability of 0.5 (50/100); will be an 'a' with probability 0.2 (20/100); and will be an 'o' with probability 0.3 (30/100). If 'e' is chosen, then the next bigram used to calculate random letters will be "**he**" since the last part of the old bigram is combined with the new letter to create the next bigram used in the Markov process.

In general, here's pseudo-code to generate random letters (and thus random text) using an order-k Markov model and a *training text* from which probabilities are calculated.

```
seed = random k-character substring from the training text
repeat N times to generate N random letters
  for each occurrence of seed in training text
    record the letter that follows the occurrence of seed in a list
  choose a random element of the list as the generated letter C
  print or store C
  seed = (last k-1 characters of seed) + C
```

Using this algorithm, here are a series of 200-randomly generated letters using the order-k Markov model. The training texts for these models are works by various authors - *Alice in Wonderland* by Lewis Carroll, *Hamlet* by William Shakespeare, and *History of the Decline and Fall of the Roman Empire* by Edward Gibbon.

Order-k	Carroll	Shakespeare	Gibbon
2	sup was it hin! What's thormomen the by fin, so sell barks of id Alied fing ver. Turn reth was fortabou enly roquithat, 'It ust ver. The to thing Cater, and posty do sherfuld ther. 'Hol hurn, you not	erselied him'd th spe th as go: whe to sureou kicur com then spartam. Hamle ars. No fir, my pron lat as. Hamink moth onsell hisel cric, mad It of wital tion to love re't call, his mastruer, [Exeunt of	s, aralablier, wage sion kinuednion p. Asister wascas of of hes re Gersuffemend be deary, Asion. Impary the pas Vant hiptimenstowe favolting the fougued The sin Gaugus, whostild Primantra, astrishower
3	sat againstanchook turnerved to Alice hously rate. 'A breal noness she withould been door proces, you mely shop on oldierse, or if she confusilence. 'Well, with on then that down of miling a dea and a	, And Queen her. Pring in awer.--Two that you my dar Speak of the putshe Quothe counsed beart: I belive Being mad mothe shortial,-- O hearing Delia.] Poloudly; the and the body did thour past I here a	d and Gibbon was of Aeged will. i. Cuperogream rebelin. I divioleridable from the life in gres afterwarms of his intone, note 43, 23: Shourtificession, withe peart, Lited the and a sing tory. Zositize

Part I: A Smarter Approach for Characters

Instead of scanning the training text N times to generate N random characters, you'll scan the text once to create a structure representing every possible k-gram used in an order-k Markov Model. **This is the first part of the project: creating the smarter/faster/better method.**

For example, Suppose the training text is *"bbbabbabbbbaba"* and we're using an order-3 Markov Model.

The 3-letter string (3-gram) *"bbb"* occurs three times, twice followed by 'a' and once by 'b'. The 3-letter string *"bba"* occurs three times, each time followed by 'b'. The 3-letter string *"bab"* occurs three times, followed twice by 'b' and once by 'a'. However, we treat the original string/training-text as circular, i.e., the end of the string is followed by the beginning of the string. This means *"bab"* also occurs at the end of the string (last two characters followed by the first character) again followed by 'b'. Other 3-grams that occur are *"abb"* and *"aba"*.

If we process the data from left-to-right, considering every 3-letter sequence and its successor character, we can efficiently (in T time) build a map of each possible three gram to the characters that follow it:

3-gram	Following characters
bbb	a, b, a
bba	b, b, b
bab	b, b, a, b
abb	a, b, b
aba	b

In your code you'll replace the brute-force re-scanning algorithm for generating random text based on characters with code that builds a map as above. Each different k-gram in the training text will be a key in the map. The value associated with a k-gram *key* is a vector of every character that follows *key* in the training text.

Part II: Markov Models for Words

This is the second part of the project. You'll use the character-generating Markov code you wrote as a starting point to create a new class that generates word models: instead of generating random characters based on the preceding character k-grams your program will generate random words based on the preceding word k-grams. Your new class should use the smarter approach using a map, not the brute-force approach!

In the *k*-order Markov model with letters you just coded, *k* characters are used to predict/generate another character. In a *k*-order *word Markov model* k words are used to predict/generate another word --- words replace characters. The idea is that whereas in your previous program you mapped every *k*-gram represented by a string to a vector of following chars, in this new program the key in the map will be some representation of a word *k*-gram. The associated value will be a list of the words that follow. You will need to decide how to represent your word *k*-grams - keep in mind that whatever you use or create must be usable as a key for a map. Fortunately, it turns out that the Standard Template Library defines the < operator for vectors (assuming the stored type is also comparable), so a vector of strings could work.

Implementation Details

You are given code that implements a working user interface for the Markov program, as well as code implementing the brute-force model. This code should compile and run as given, letting you test with the brute-force model. You need to provide header files and source files for the models you create in parts I and II. You will also need to edit two places in markov.cpp, marked with a TODO comment to activate your new model code.

Testing

This project can be difficult to test, given the random nature of the output. For part I, unless you've taken a very different approach from the expected (which is fine), your map_model should reproduce the same output as the brute_model when given the same starting random seed. (That is, try setting the random seed to a specific value, like 1234, each time you run the brute or map-based model. If you get the same text, then your map_model is probably correct.) This won't work on the word_model, of course!

Besides that, the best guidance we can offer is that, the lower the order, the less sense your output should make. Conversely, if you set the order very high (say, 20 for the character-based models, or 5 for the word-based model), you should start quoting fairly large chunks of the original text.