Ngrams!

It is a great deal of surprise. That she should suffer but myself?

CSCI 141, Project #3, Fall 2013

Due date: Submit on Canvas by Friday, December 6.

Random English: If we want to generate random English text we could produce a random string of words. But this wouldn't look much like English:

> fishery witness roaster wrestle accredit reclamation revised dialectic amoebae

Sampling words proportionally to their occurrence in actual English would help, but we can do even better than that. One reason is that certain words more probably follow other words. If we use a some known English as a model we can use it to predict (given a few starting words) what word comes next. For example, if we look at the following sample of excellent English:

> I will not eat them in a box.
> I will not eat them with a fox.
> I will not eat them on a train.
> I will not eat them in the rain.

We can see that following `I will not eat them` we have three choices, `in`, `with`, or `on`. Other words, like `a` or `fox` are not probable at all. Further, we can see that `in` happens 2 out of 4 times, so it is twice as probable as either `with` or `on`.

If we use a very large corpus, like an entire novel, we can, given any string of 2 or 3 words, find out what words are likely to come next. Generating words in *probable* order may result in better looking random English.

The Ngram Application: An ngram is a string of n words. `I will not eat them` is a 5-gram. Ngrams have many important applications in processing natural language. Here we will use ngrams from a work of English to generate new "English" in the same style.

As a demonstration, let us fix the length of our ngram at 2. We start with the first two words of a novel, and then, out of all possible words that follow those two, we randomly select one. Now we have three words. We use the second two words as a new ngram, and find a third word based on the words in the book that follow these two.

For example, *Alice in Wonderland*, by Lewis Carroll, begins with

> Alice was beginning to get very tired ...

So our prefix starts with `Alice was`. We now look at all the occurrences of `Alice was` in the entire novel (there are seventeen), and look at what words follow this ngram, we get the following list:

> beginning not not soon so more just not beginning a silent. rather very too very thoroughly only

So, if we want a "logical" following word, just pick a random one from the list; for example, `thoroughly`. This gives us one more word, and a new 2-gram, `was thoroughly`. It turns out that this 2-gram only occurs once in the entire book, and is followed by `puzzled.`, so our new 2-gram is `thoroughly puzzled.` Look through the book to find what words follow `thoroughly puzzled.`, and repeat.

We can continue in this manner for as long as we like (provided we don't accidentally find the last ngram in the book). One random passage through the ngrams resulted in the following:

> Alice was thoroughly puzzled. 'Does the boots and shoes!' she repeated in a hoarse growl, 'the world would go round and round Alice, every now and then, and holding it to half-past one as long as you go to law: I will tell you his history,' As they walked off together. Alice laughed so much at this,

Implementation: First, we build a dictionary of ngrams from the book. With n fixed (2 and 3 are usually the most amusing), we read every word in the book and collect every ngram we find in a dictionary. Use a simple list for the ngram, it will make it easy to remove one word and add the next when shifting from one ngram to the next.

Unfortunately, lists cannot be used as keys in Python dictionaries. However, since these are always lists of strings, we can simply convert them into strings using `' '.join(ngram)` and use that as the dictionary key.

Now, stored under the ngram is a list of every word that follows the ngram. Finding the ngrams, and also building up the lists of words for each ngram, can all be done in a single pass through the book: every time we get a new word from the book, we store it under the ngram, and then shift the new word into the ngram. Repeat.

Note that it is a *list* of words stored under each ngram. The first time you see the ngram, you will have to create a new list to store in the dictionary. On subsequent encounters of this ngram, you will append the new word to the list of words already there.

Now, dictionary in hand, we can proceed to generate text. We start with the first two words in the novel as the first ngram. We look up this ngram in the dictionary, which gives us a list of words following that ngram. Pick a random one from this list, giving the next ngram, and continue until we have generated as many words as we like. If you should be so unlucky as to hit the last ngram in the book, there may be no words following it, so be sure to guard against this eventuality. (You can test by using a very short text.)

For the purposes of this assignment, a "word" is a string of text separated by whitespace (spaces, tabs, newlines, and any control characters in the text). In the example above, the word `puzzled.` includes the period. The Python `split` procedure will give you words of this kind. To read the entire book into a string, just use the process from the last project.

Note: it is slightly easier to program if you assume each book starts with a n "nonwords" like `XXXXX`. Then you don't have to find the first ngram in the book when you start reading or generating, you just preload the prefix with n nonwords. (Just don't print the nonwords when generating!) This is optional.

Find the text of one of your favorite novels, documents, whatever (don't violate any copyrights!) and generate your own garbage in the style of a great writer! Our subtitle comes from Jane Austen! When submitting, zip your program files together with your sample text and make it so 100 words of amusing text is generated by simply pressing the run button.