Contents

1 Generating sentences with Markov models

The aim of this week's practical class is to experiment with the Markov processes for generating text. You will write some code (in Java, or any other language of your choice, using whatever IDE you are happy with) to process a source text, then generate a **probability transition table**, then look up words in that table to generate new sentences.

1.1 Constructing the model

Create a new class (call it, e.g., `Markov`) and create a method that will be the main method of the class. In my version of the code I wrote all of the code in this single method, and I will describe this version as we go, but you can use multiple methods if you prefer.

1.2 Preparing the data for the model

1.2.1 Resources

Start by downloading the short file called `testFile.txt` from the Moodle page. You can use this whilst you are developing your program, and then try it with a longer file once you get it working. The sentences in the test file are deliberately repetitive and use a small vocabulary.

1.2.2 Tokenize the string

Create a `String` called content and read the entire content of `testFile.txt` into it. Use string processing methods to strip out everything apart from letters and spaces, and turn all of the words into lower case. Then, create an array-like structure (e.g. an array or `ArrayList`) called words and put each word into words in the same order as in the text file.

1.3 Prepare the model

1.3.1 Assign each word a number

Create two maps (e.g. `HashMap`), one from `String` to `Integer`, and one the other way round. Write a loop to iterate through words, and each time you come across a new word, give it a new number. Add these pairs to the maps. So you should now have a map that looks like this:

```
cat 1
sat 2
mat 4
on 3
the 0
where 6
was 5
```

... and one the other way

```
0 the
1 cat
2 sat
3 on
4 mat
5 was
6 where
```

(If you want, you could write a data structure entirely indexed by strings; but, writing the two dimensional array indexed by strings is a little complex in Java, so perhaps this is the best way to do it).

1.3.2 Count each of the unique words

Count the number of distinct words (the number of entries in one of the maps—they will, of course, both be of the same size), and call this variable d (for distinctive words).

1.3.3 Create a d-by-d array

Now, create a d-by-d array of integers called count. This is going to contain a count of how often one word is followed by another.

1.3.4 Count each time a pair of words occurs

Iterate through the text, looking at (overlapping) pairs of words: "the cat", "cat sat", "sat on" etc. (these are what we called bigrams in the lecture). Each time you find a pair, look up the index of the two words (call these `firstWord` and `secondword`) and increment the value of `count[firstWord][secondWord]` by 1. So, once you have been through the text, you should have a matrix like this (`firstWord` running vertically, `secondWord` horizontally):

```
0 4 0 0 4 0 0
0 0 4 0 0 0 0
1 0 0 3 0 0 0
3 0 0 0 0 0 0
2 0 0 0 0 1 0
0 0 0 0 0 0 1
1 0 0 0 0 0 0
```

Print count to check that this is accurate. We can read this as follows. In the text, the word "the" (index 0, so the first row) is followed 4 times by the word "cat" (index 1), 4 times by the word "mat" (index 4), and no times by any other word. Now, create another array to represent the total across the row:

```
0 4 0 0 4 0 0 8
0 0 4 0 0 0 0 4
1 0 0 3 0 0 0 4
3 0 0 0 0 0 0 3
2 0 0 0 0 1 0 3
0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 1
```

Create a d-by-(d+1) array called prob. This will be used to store the probability that, in the source text, of each word following the other word. Actually, this will be a cumulative probability matrix, to help us pick a value later on. This is the most complex part of the process, so be careful here. Initialise the first column to all zeros, then write a loop that iterates across

each row of the matrix, adding on to the value in the previous column of prob (important!) the value at that point in the count array divided by the total for that row. If you print this out, it will look something like this.

```
      the  cat  sat  on   mat  was  where
the   0.00 0.00 0.50 0.50 0.50 1.00 1.00  1.00
cat   0.00 0.00 0.00 1.00 1.00 1.00 1.00  1.00
sat   0.00 0.25 0.25 0.25 1.00 1.00 1.00  1.00
on    0.00 1.00 1.00 1.00 1.00 1.00 1.00  1.00
mat   0.00 0.67 0.67 0.67 0.67 0.67 1.00  1.00
was   0.00 0.00 0.00 0.00 0.00 0.00 0.00  1.00
where 0.00 1.00 1.00 1.00 1.00 1.00 1.00  1.00
```

How to read this? Well, the top row represents bigrams beginning with the word "the". Imagine that we pick a random number in the range [0.0,1.0)—e.g. imagine that we pick the number 0.76. Now, we work along the row, until we find the first pair of values that our random number sits between. 0.76 isn't between 0.00 and 0.00 (nothing is!), so we don't follow the word "the" with the word "the". Move along. 0.76 isn't between 0.00 and 0.50, so this time we don't choose the word "cat"—though, if the random number had been, say, 0.13, then we would have done. We move along until we reach the pair 0.50, 1.00—now, 0.76 is between those, so we choose the corresponding word: "mat". Implement this process. Start with a starting word to seed the process, for example "the", then loop around for a fixed number of words, choosing each word based on the previous word according to the probability table, and printing out the words found. You should be able to generate odd but not totally ungrammatical sentences like "the mat was where the cat sat on the cat".

1.4 Extensions

1. Scale this up so that it uses a much larger text as the source, for example a novel from the full-text collections at `http://www.gutenberg.org` or the "baby" sample (just 4 million words!) from the British National Corpus at `http://www.natcorp.ox.ac.uk` To do this you will probably need to change the representation of the matrix to use the idea of a sparse matrix, i.e. one where most of the entries are zero, and so you only store the entries where there is something—otherwise the memory requirements are huge.

2. Decide when to stop, automatically. You will need to do something with punctuation, for example turning the full stop into an end-of-sentence marker and adding it into the matrix.

3. Don't just look at bigrams, look at trigrams: your matrix is indexed not just by the previous word but by the previous two words. Now, you really will have to use sparse matrices at this point.

4. Implement similar ideas, but for music. The words are notes, the source "text" a set of tunes, etc.