Practical 1: A Markov Ulysses

James Joyce's *Ulysses* is described as anything from the greatest novel in the English language to unintelligible gibberish, depending on who you ask. The less positive readers tend to argue that the stream of consciousness in which the book is largely written just appears random. One way to investigate that claim is to generate random sequences of text based on the word patterns in *Ulysses* to see how easily the real and random text can be distinguished. At some level, of course, generating random text based on the patterns found in human written text is exactly what large language models do, training very complex models on huge volumes of text. In this practical you will instead use what you might call a 'small language model', based on a single text.

The idea is to use a second order Markov model — a model in which we generate words sequentially, with the each word being drawn with a probability dependent on the two words preceding it. The probabilities are obtained from the actual text, by simply tabulating the frequency with which each word follows any other pair of words.

To make this work requires some simplification. The model will not cover every word used in the text. Rather the model's 'vocabulary' will be limited to the m most common words. $m \approx 1000$ is sensible. Suppose that the m most common words are in a vector $\mathbf{b}$. Let $\mathbf{a}$ be the vector of all words in *Ulysses*. We need to know the probabilities of a_t being each of the words, b_j, in $\mathbf{b}$, given it is in $\mathbf{b}$ at all, and that $a_{t-1} = b_k$ and $a_{t-2} = b_i$:

$$\text{P}(a_t = b_j | a_{t-1} = b_k, a_{t-2} = b_i, a_t \in \mathbf{b})$$

One way to do this would be to go through the text simply counting up how often b_j follows the word pair b_i, b_k, storing the results in an $m \times m \times m$ array A so that the estimate of the probability given above would be $\texttt{A[,i,k]/sum(A[,i,k])}$ whenever $\texttt{A[,i,k]}$ is non-zero. Given A we could then generate sequences of words, based on these probabilities, given a pair of starting words. The problem is that if $m \approx 1000$ then A would require around 8Gb of storage, most of which would be full of zeros (all the common word triplets that never occur).

An alternative computational approach to using this model is to store all the common word triplets in the text in a 3 column matrix. Then each time we need $\text{P}(a_t = b_j | a_{t-1} = b_k, a_{t-2} = b_i, a_t \in \mathbf{b})$ we simply select all the rows of the matrix that contain both b_i in the first column and b_k in the second column. From those rows we can tabulate the third column to get the probabilities of each word in $\mathbf{b}$ being the third word of the triplet. Computing the probabilities *on the fly* in this way costs a bit more computer time than computing them in advance, but uses far less computer memory. At most the approach requires storage for the number of words in the text multiplied by 3 - about 8Mb - but actually the number of common word triplets will be far fewer than the number of words, so the storage requirement is much less still.

There are some fussy details needed in practice.

1. It is possible that some pair of words from $\mathbf{b}$ is *never* followed by a word also in $\mathbf{b}$. In that case the model should generate the next word from just the single previous word according to the probability

$$\text{P}(a_t = b_j | a_{t-1} = b_i, a_t \in \mathbf{b}).$$

   This probability can be computed on the fly, in the same way as was done for word triplets, if we store all the common word pairs in a two column matrix.

2. ...but it is also possible that a single word in $\mathbf{b}$ is never followed by another word in $\mathbf{b}$. In that case the next word should be generated from

$$\text{P}(a_t = b_i | a_t \in \mathbf{b})$$

   which can be obtained by just counting up the number of times that each b_i occurs in the text

3. To start generating words from the model, we might randomly pick a word from $\mathbf{b}$ as in 2 and then pick the next word as in 1. Given the starting pair, the full model can be run.

Because this is the first practical, the instructions for how to produce code will be unusually detailed: the task has been broken down for you. Obviously the process of breaking down a task into constituent parts before coding is part of programming, so in future practicals you should expect less of this detailed specification.

As a group of 3 you should aim to produce well commented[1], clear and efficient code for training the model and simulating short sections of text using it. The code should be written in a plain text file called `proj1.r`

[1]Good comments give an overview of what the code does, as well as line-by-line information to help the reader understand the code. Generally the code file should start with an overview of what the code in that file is about, and a high level outline of what it is doing. Similarly each function should start with a description of its inputs outputs and purpose plus a brief outline of how it works. Line-by-line comments aim to make the code easier to understand in detail.

and is what you will submit. Your solutions should use only the functions available in base R. The work must be completed in your work group of 3, which you must have arranged and registered on Learn. The first comment in your code should list the names and university user names of each member of the group. The second comment **must** give a brief description of what each team member contributed to the project, and roughly what proportion of the work was undertaken by each team member. Contributions never end up completely equal, but you should aim for rough equality, with team members each making sure to 'pull their weight', as well as not unfairly dominating[2].

1. Create a repo for this project on github, and clone to your local machines.

2. Download the text as plain text from `https://www.gutenberg.org/files/4300/4300-0.txt`.

3. The following code will read the file into R. You will need to change the path in the `setwd` call to point to your local repo. Only use the given file name for the Ulysses text file, to facilitate marking.

   ```
   setwd("put/your/local/repo/location/here") ## comment out of submitted
   a <- scan("4300-0.txt",what="character",skip=73,nlines=32858-73)
   a <- gsub("_(","",a,fixed=TRUE) ## remove "_("
   ```

   Check the help file for any function whose purpose you are unclear of. The read in code gets rid of text you don't want at the start and end of the file. Check out what is in `a`. The `setwd` line should be commented out of the code you finally submit for marking.

4. Some pre-processing of `a` is needed. Write a function, called `split_punct`, which takes a vector of words as input along with a punctuation mark (e.g. `","`, `"."` etc.). The function should search for each word containing the punctuation mark, remove it from the word, and add the mark as a new entry in the vector of words, after the word it came from. The updated vector should be what the function returns. For example, if looking for commas, then input vector

   ```
   "An" "omnishambles," "in" "a" "headless" "chicken" "factory"
   ```

   should become output vector

   ```
   "An" "omnishambles" "," "in" "a" "headless" "chicken" "factory"
   ```

   Functions `grep`, `rep` and `gsub` are the ones to use for this task. Beware that some punctuation marks are special characters in regular expressions, which `grep` and `gsub` can interpret. The notes tell you how to deal with this.

5. Use your `split_punct` function to separate the punctuation marks, `","`, `"."`, `";"`, `"!"`, `":"` and `"?"` from words they are attached to in the text.

6. The function `unique` can be used to find the vector, `b`, of unique words in the Ulysses text, `a`. The function `match` can then be used to find the index of which element in `b` each element of `a` corresponds to. Here's a small example illustrating `match`.

   ```
   match(c("tum","tee","tum","tee","tumpty","tum","wibble","wobble"),c("tum","tee"))
   [1]  1  2  1  2 NA  1 NA NA
   ```

   (a) Use `unique` to find the vector of unique words. Do this having replaced the capital letters in words with lower case letters using function `tolower`.

   (b) Use `match` to find the vector of indices indicating which element in the unique word vector each element in the (lower case) text corresponds to (the index vector should be the same length as the text vector `a`).

   (c) Using the index vector and the `tabulate` function, count up how many time each unique word occurs in the text.

[2]all team members must have git installed and use it - not doing so will count against you if there are problems of seriously unequal contributions

(d) You need to decide on a threshold number of occurrences at which a word should be included in the set of $m \approx 1000$ most common words. Write code to search for the threshold required to retain ≈ 1000 words.

(e) Hence create a vector, `b`, of the m most commonly occurring words.

7. Now you need to make the matrices of common word triplets and pairs (`T` and `P`, say).

   (a) Use `match` again to create a vector giving which element of your most common word vector, `b`, each element of the full text vector corresponds to. If a word is not in `b`, then `match` gives an `NA` for that word (don't forget to work on the lower case Ulysses text).

   (b) Now create a three column matrix (e.g. using `cbind`), in which the first column is the index of common words, and the next column is the index for the following word. i.e. the index vector created by `match` followed by that vector shifted by one place. The final column should be the index vector for the words shifted one more place again. You need to remove a couple of entries from the start and/or end of each vector as appropriate). Each row of your matrix indexes a triplet of adjacent words in the text. When a row has no `NA`s then we have a triplet of common words, which will contribute to our `T` array.

   (c) Using `rowSums` and `is.na` identify the common word triplets, and drop the other word triplets (those that contain an `NA`).

   (d) Using the same ideas, produce the two column common word pairs matrix, `P`.

8. Finally write code to simulate 50-word sections from your model. Do this by using the model to simulate integers indexing words in the word vector `b`. Then print out the corresponding text with `cat`. The `sample` function should be used to select a word (index) with a given probability. The simplest way to generate words with the correct probabilities does the following

   (a) Suppose the last generated pair of words were indexed `k[i]` and `k[j]`.

   (b) Extract the sub-matrix from `T` whose rows all have `k[i]` in column 1 and `k[j]` in column 2.

   (c) Pick 1 element at random from the third column of your extracted sub-matrix, using the `sample` function.

   Make sure you understand why this picks the next word with the correct probability (and why we didn't need to actually tabulate the third column of the sub-matrix). Note that if the sub-matrix has no rows, then you will need to simulate the next word using the pairs matrix, `P`, instead (and if that doesn't work just simulate according to the common word frequencies).

9. For comparison, simulate 50 word sections of text where the word probabilities are simply based on the common word frequencies, with each word drawn independently.

10. If you get everything working and have time to go for the last 3 marks, then modify your code so that words that most often start with a capital letter in the main text, also start with a capital letter in your simulation. But do make sure that you achieve this in a way that does not mess up the word frequencies! Hint: think about a modified version of `b` for printing.

One piece of work - the text file containing your commented R code - is to be submitted for each group of 3 on Learn by 12:00 6th October 2023. You may be asked to supply an invitation to your github repo, so ensure this is in good order. No extensions are available on this course, because of the frequency with which work has to be submitted. So late work will automatically attract a penalty (of 100% after work has been marked and returned). Technology failures will not be accepted as a reason for lateness (unless it is provably the case that Learn was unavailable for an extended period), so aim to submit ahead of time.

Marking Scheme: Full marks will be obtained for code that:

1. does what it is supposed to do, and has been coded in R approximately as indicated (that is marks will be lost for simply finding a package or online code that simplifies the task for you).

2. is carefully commented, so that someone reviewing the code can easily tell exactly what it is for, what it is doing and how it is doing it without having read this sheet, or knowing anything else about the code. Note that *easily tell* implies that the comments must also be as clear and *concise* as possible. You should assume that the reader knows basic R, but not that they know exactly what every function in R does.
3. is well structured and laid out, so that the code itself, and its underlying logic, are easy to follow.
4. is reasonably efficient. As a rough guide the whole code should take a few seconds to run - much longer than that and something is probably wrong.
5. includes the final part - but this is only worth 3 marks out of 18.
6. was prepared collaboratively using git and github in a group of 3.
7. contains no evidence of having been copied, in whole or in part, from other students on this course, students at other universities (there are now tools to help detect this, including internationally), online sources etc.
8. includes the comment stating team member contributions.

Individual marks may be adjusted within groups if contributions are widely different.