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. 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 an pth-order Markov model — a model in which we generate words sequentially, with each word being drawn with a probability dependent on the p words preceding it. The probabilities are obtained from the actual text, by simply finding the set of words that follow any particular sequence of p words and tabulating their frequency. For example, consider the word pair "I am". Working through the actual text, we might find that this is followed by "a" 5 times, "tired" twice, "cold" twice and "God" once, so the probabilities of each of these words occurring next are 0.5, 0.2, 0.2 and 0.1, for the 2nd order model.

To make this work well requires some simplification. The model will not cover every word used in the text, since for rare words we have too little data to get good estimates of the probabilities. This will lead to unconvincing text being generated (similar to 'hallucination' in large language models). Rather the model's 'vocabulary' will be limited to the m most common words. $m \approx 1000$ is sensible.

You could store the estimated probabilities in an array. For example for a 2nd order model we might use an $m \times m \times m$ array, to store the probabilities of each common word following each possible pair of common words, estimated from the text. But this is expensive in terms of computer memory. For $m = 1000$ it requires 8Gb of storage and 1000 times that again for a 3rd order model. It is also wasteful. Most length m word sequences will never occur in the actual text, so we would store a huge number of zero probabilities.

Instead, if n is the number of words in the text, we can create a matrix, $\mathbf{M}$ with $n - p$ rows and $p + 1$ columns, where the rows contain all the sequences of $p + 1$ words occurring in the text. Rather than store the actual words, we will store 'tokens' representing the words. Here the token will be the number giving the position of the word in a vector, $\mathbf{b}$, of the m most common words, and an `NA` for words not in that list. Storing tokens is more memory efficient than storing whole words.

Then let **w** be a p-word sequence of common word tokens. We need to find all rows of $\mathbf{M}$ that start with this **w** and pick randomly from the tokens occurring as the $p + 1$th elements of these rows, excluding any `NA`s. This approach picks the next word with the desired probability. We proceed iteratively. The most recently generated word token is appended to the end of **w**, while the first word token is dropped, to get the **w** for the next step.

What should happen if there is only one option for the next word, or only NA tokens, or no rows matching $\mathbf{w}$? In that case try the $p - 1$th order model (the first p columns of $\mathbf{M}$ already contain the required information). If that fails try the $p - 2$th order model and so on. The 0th order model simply picks a non-`NA` token at random from the first column of $\mathbf{M}$.

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` 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.

[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. The comments should not be mechanical descriptions of what lines of R code do. The aim is to explain what is being done and why, in a way that helps the reader understand the logic and purpose behind the code.

[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

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,
             fileEncoding="UTF-8")
   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. The idea is that the model will treat punctuation just like words.

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 upper case 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.

   (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 token sequences. Let `mlag` be the maximum lag considered (i.e. p above). Your code should be written to work with any reasonable value of `mlag`, but set it to 4 to start with. You need to construct the matrix $\mathbf{M}$.

(a) Use `match` again to create a vector of the same length as `a` 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 an (`n - mlag`) × (`mlag + 1`) matrix, `M`, 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 next column is shifted once more again, and so on. You will need to remove entries from the start and/or end of each shifted index vector as appropriate. Each row of `M` gives the indices in `b` of a sequence of `mlag+1` adjacent words in the text.

8. Finally write code to simulate `nw`-word sections from your model (set `nw` to 50 for submission). Do this by using the model to simulate tokens: integers indexing words in vector `b`. Then print out the corresponding text with `cat`. The `sample` function should be used to select a word token with a given probability. Generate the first token by sampling randomly from the vector of all non-NA tokens for the whole text. The simulation will involve nested loops, something like the following, where you need to fill in the `...`.

```
for (i in 2:nw) {
  for (j in mlag:1) if (i>j) { ## skip lags too long for current i
  ...
  }
}
```

Your code should `break` out of the `j` loop as soon as the next word has successfully been generated.

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. Hint: this only requires a modified version of `b` for printing. To make things even nicer, you could further modify your code so that there is no space before punctuation (think about whether spaces should be produced by `cat` or be attached to words in your modified `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 (noon) 4th October 2024. 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 anyone else (AI counts as anyone else for this purpose), other students on this course, students at other universities (there are now tools to help detect this, including internationally), online sources, ChatGPT etc.
8. includes the comment stating team member contributions.

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