Practical 1: A Markov Text Model

New translations of the bible into modern English always arouse controversy. Proponents of the new translation argue that it makes the text more understandable, while opponents argue that the new text merely removes the rhythm and poetry of the language while making it no more accessible. There are various ways of testing claims about understandability. One test is to see how readily readers can distinguish genuine text from the book, with computer generated text designed to match word patterns seen in the book, but generated randomly so that it contains no actual meaning.

This practical is about creating such computer generated text. The idea is to use a 2nd order Markov model — that is a model in which we generate words sequentially, with the each word being drawn with a probability dependent on the words preceding it. The probabilities are obtained by training the model on the actual text. That is by simply tabulating the frequency with which each word follows any other pair of words.

To make this work requires 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. $m \approx 500$ is sensible. Suppose that the m most common words are in a vector $\mathbf{b}$. Let a be the vector of all words in the Bible. We will construct an $m \times m \times m$ array $\mathbf{A}$, such that

$$\mathrm{P}(a_t = b_j | a_{t-1} = b_k, a_{t-2} = b_i) = T_{ikj}.$$

Given $\mathbf{T}$, $\mathbf{b}$ and a pair of starting words from $\mathbf{b}$, you can then iterate to generate text from the model. That is, given a word b_i followed by a word b_k, the following word has probability T_{ikj} of being b_j. To generate an appropriate b_j, we use the `sample` function to sample a word from $\mathbf{b}$ with probabilities given by `T[i,k,]`. To estimate $\mathbf{T}$ you need to go through the text of the bible, counting up the number of times b_j follows sequential word pairs b_i, b_k for all words in $\mathbf{b}$.

There are some fussy details needed in practice.

1. It is possible that some word pair in $\mathbf{b}$ is *never* followed by a word also in $\mathbf{b}$, so we end up with zero estimated probability for any word as the next word! In that case the model should generate the next word from just the single previous word according to the probability

   $$\mathrm{P}(a_t = b_j | a_{t-1} = b_i) = A_{ij}.$$

   $\mathbf{A}$ can be obtained by counting up each time b_j follows b_i in the text.

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

   $$\mathrm{P}(a_t = b_i) = S_i$$

   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 $\mathbf{b}$, based on the probabilities in $\mathbf{S}$, and then pick a second word based on that word and the probabilities in $\mathbf{A}$ (falling back on $\mathbf{S}$ as needed). Once we have the starting pair, more words are simulated using the full model.

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] Good comments give and 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.

[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

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/ebooks/10`.

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 bible text file, to facilitate marking.

```
setwd("put/your/local/repo/location/here")
a <- scan("pg10.txt",what="character",skip=104) ## skip contents
n <- length(a)
a <- a[-((n-2886):n)] ## strip license
a <- a[-grep("[0123456789]:[0123456789]",a)] ## strip out verse numbers
```

   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`.

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 bible text.

6. The function `unique` can be used to find the vector, `b`, of unique words in the bible 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 indicies indicating which element in the unique word vector each element in the (lower case) bible text corresponds to (the index vector should be the same length as the bible 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 $m \approx 500$ most common words. Write code to search for the threshold required to retain ≈ 500 words.

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

7. Now you need to make the **T** array.

   (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 bible 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 $\mathbf{T}$ array.

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

(d) Now loop through the common word triplets adding a 1 to `T[i,k,j]` every time the jth common word follows the pair `i,k`. Make sure `T` is initialized to the right thing before you start counting.

(e) In principle `T` should be standardized for its entries to be interpreted as probabilities, but this can be skipped. You only need `T` for supplying sampling probabilities to `sample`, and `sample` only requires vectors of probabilities to be supplied up to a normalizing constant - it will standardize internally.

(f) Now produce matrix $\mathbf{A}$ and vector $\mathbf{S}$ using a similar approach to that used for $\mathbf{T}$.

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.

9. For comparison, simulate 50 word sections of text where the word probabilities are simply taken from $\mathbf{S}$.

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 Friday 7th October 2022. You may be asked to supply us with 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 less than a minute 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.