Practical 1: A Shakespeare sentence simulator

Large language models, such as DeepSeek and chatGPT, are trained on huge quantities of text to predict the likely responses to user queries and prompts. This practical is about coding up what you might term a simple 'small language model' to simulate sentences resembling those written by Shakespeare.

We will work with a Markov model. Given a sequence of p words, the model will randomly select the next word depending on the probabilities that each possible word follows the given sequence. These probabilities are computed from the text of the complete works of Shakespeare. Typically we would start with a single word, generate the next word from the model and then use the resulting 2 word sequence to generate the next word, then the 3 word sequence to generate the next word until we have an m word sequence, where m is the maximum order of model considered. Given an m word sequence we then predict the next word. To predict each word thereafter, we use the sequence of the most recent m words generated.

How exactly should these following-word probabilities be obtained from the text? For each m word sequence we could simply locate each time the sequence occurs in the text and tabulate the words that follow it, and their frequency. For example, consider the ($m = 2$) 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 (with the probabilities for all other words set to 0).

How large should m be? There is a trade-off here. Using a large m gives lots of 'context' from which to predict the next word, but typically leads to only a small (or no) occurrences of the word sequence in the text - so we get a rather poor noisy estimate of the likely distribution of following words. One way to deal with this is to predict the next word from a mixture of next word distributions generated my the last m words, the last $m - 1$ words and so on down to the last word. This gives a nice balance between the richer context provided by larger word sequences and the richer data available for shorter ones.

To be more mathematically concrete, let $\mathbf{v}$ be the vector containing the latest sequence of m words. Then we generate the next word from the distribution

$$\sum_{i=1}^{m} w_i P(\text{next word}|\mathbf{v}[i:m]) \tag{1}$$

where the w_i are mixture weights summing to 1 ($w_i = 1/m$ by default).

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 k most common words. $k \approx 1000$ is sensible. Secondly, we will not work with the raw words, but with numerical 'tokens' representing them. The tokens will be the integers giving the location of the word in the vector of k most common words (or `NA` is the word is not in the common word set.) For example, if $\mathbf{b}$ is the vector of most common words and $b_9 =$"forsooth" then the token for "forsooth" is 9.

How best to store the probabilities in (1) and sample words/tokens according to those probabilities? The obvious approach creates a $k \times k \times \cdots \times k$ array ($m+1$ dimensions) containing the probabilities of each common word token following each possible common word token sequence of length m. We then work through the text adding one to the array entry indexed by each length $m+1$ token sequence encountered. At the end we normalize the entries sharing the same initial m tokens (where these sum to > 0) to obtain the probability estimates[1]. The problem is that this approach would require 8 terabytes of storage for $m = 4$ and $k = 1000$. Meanwhile most of the array entries would be zero as most word combinations never occur. All far too inefficient to use.

Instead, if n is the number of words in the text, we can create a matrix, $\mathbf{M}$ with $n - m$ rows and $m + 1$ columns, where the rows contain all the sequences of $m + 1$ word tokens occurring in the text. This will let us sample according to the model probabilities 'on the fly'. Here is how...

Let $\mathbf{v}$ be an m word token sequence. We first find all the rows of $\mathbf{M}$ whose first m elements match $\mathbf{v}$, and whose $m + 1$th element is not `NA`. Now collect the $m + 1$th element of each of those rows into one vector $\mathbf{u}$, and associate with each element of $\mathbf{u}$ a probability w_m/n_u where n_u is the number of elements in $\mathbf{u}$ and w_m the mixture weight. Together $\mathbf{u}$ and the probability constitute an empirical representation of the distribution of words that might follow $\mathbf{v}$ in the text. Typically words may occur in $\mathbf{u}$ several times.

We can now drop the first element of $\mathbf{v}$ and the first column of $\mathbf{M}$ and repeat the process to get the distribution of words likely to follow `v[2:m]`. Similarly the distributions of words likely to follow `v[3:m]` etc can be obtained.

[1] we would need to repeat this for $m - 1$, $m - 2$ etc order models

The combined **u** vectors and corresponding probabilities then represent an approximation to the distribution in (1). We can sample from this distribution by simply concatenating the **u** vectors, and the corresponding vectors of probabilities, and using the `sample` function to sample one token from this distribution. In the event that the combined **u** vector contains no elements, we may as well select a common word at random from the text (i.e. sample a common word according to its occurrence probability in the text).

Your task is to write code implementing this model and simulating sentences from it. 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, collaborating using `git`, you should aim to produce well commented[2], clear and efficient code implementing the model and simulating sentences with 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[3].

Note that `unlist` can me quite useful for some of the tasks below, and that you will almost certainly progress much faster if you bother to learn how to use the `debug` package to `mtrace` your functions.

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/100.txt.utf-8`.

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 Shakespeare text file, to facilitate marking.

   ```
   setwd("put/your/local/repo/location/here") ## comment out of submitted
   a <- scan("shakespeare.txt",what="character",skip=83,nlines=196043-83,
             fileEncoding="UTF-8")
   ```

   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.

   (a) The text contains stage directions, starting `[` and ending `]`, but with one or two unmatched brackets. These stage directions need to be removed. A reasonable strategy is to locate all words in `a` that contain `[` using `grep`. Then loop through the identified word locations in `a` using `grep` again to search for the corresponding `]` within the next 100 words. In this way you can build a record of all the words corresponding to stage directions. Once this record is assembled, use it to delete the stage directions.

   (b) Words that are fully upper case are character names indicating who is speaking, or headings of various sorts. Similarly, numbers expressed as arabic numerals are not part of the text. All should be removed. You can identify the fully upper case words and numerals by testing whether words are equal to their upper case version produced by `toupper`. Note that "I" and "A" are special cases which should not be removed!

   (c) Using `gsub` you should remove "_" and "-" from words (the last one is debatable actually).

[2]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.

[3]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) Write a function, called `split_punct`, which takes a vector of words as input along with a vector of punctuation marks (e.g. `","`, `"."` etc.). The function should search for each word containing the punctuation marks, remove them 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 the 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.

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

(f) For simplicity convert your cleaned word vector `a` to lower case, which is what will be used from here on.

5. The function `unique` can be used to find the vector, `b`, of unique words in the (cleaned) 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.

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

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

   (d) Create a vector `b` containing the ≈ 1000 most common words. The `rank` function is very useful.

6. Now you need to make the matrices of common word token sequences. Let `mlag` be the maximum lag considered (i.e. m 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. So this vector contains the tokens for representing the whole text.

   (b) Now create an (`n - mlag`) $\times$ (`mlag + 1`) matrix, `M`, in which the first column is the token vector, and the next column is the token for each 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 token vector as appropriate. Each row of `M` gives the indices in `b` (tokens) of a sequence of `mlag+1` adjacent words in the text.

7. Now write a function

```
next.word(key,M,M1,w=rep(1,ncol(M)-1))
```

where `key` is the word sequence for which the next word is to be generated, `M` is as defined above, `M1` is the vector of word tokens for the whole text and `w` is the vector of mixture weights (which actually don't need to be normalized). The function should return a token for the next word, generated according to the model described above. It should be able to deal appropriately with any length of `key`: using reduced order versions of the model for short keys, and using only data from the end of `key` if it is too long.

The crucial part of the function is (repeatedly) finding the rows of `M` that match `key` (or its reduced versions). Suppose the current `key` is to be matched to columns `mc:mlag` of `M`. Now compute

```
ii <- colSums(!(t(M[,mc:mlag,drop=FALSE])==key))
```

If `ii[j]`=0 and is finite (see `?is.finite`) then row `j` of `M` contains a match.

8. Now write code to select a single word token (but not punctuation) at random from the text, to use to start simulating from the model. Alternatively, you might like to find the token for an interesting word like 'romeo' from the text and use that as the starting point.

9. Finally write code to simulate from the model until a full stop is reached. Then convert the generated tokens back to words and print them nicely (using `paste` for example.)

You might like to compare the results to 'sentences' obtained by simply drawing common words at random from the text until a full stop is drawn.

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) 3rd October 2025. 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 using base R (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 at most a few 10s of seconds to run - much longer than that and something is probably wrong.

5. was prepared collaboratively using git and github in a group of 3.

6. 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.

7. includes the comment stating team member contributions.

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