Guide to the ngram Package

An n-gram Babbler

Drew Schmidt

Guide to the **ngram** Package

An n-gram Babbler

June 17, 2014

Drew Schmidt
wrathematics@gmail.com

Version 0.1-0

Cover art is *Hydra*, uploaded to openclipart.org by Tavin.

This publication was typeset using LaTeX.

Contents

1 Introduction

2 Installation

In this section, we will describe the various ways that one can install the **ngram** package.

2.1 Installing from Source

The sourcecode for this package is available (and actively maintained) on GitHub. To install this (or any other) package from source on Windows, you will need to first install the Rtools package.

The easiest way to install **ngram** from GitHub is via the **devtools** package by Hadley Wickham. To install **ngram** using **devtools**, simply issue the command:

```
library(devtools)
install_github(repo="ngram", username="wrathematics")
```

from R. Alternatively, you could download the sourcecode from github, unzip this archive, and issue the command:

```
R CMD INSTALL ngram-master
```

from your shell.

2.2 Installing from CRAN

The usual

```
install.packages("ngram")
```

from an R session should do it.

3 Using the Package

3.1 Background

The canonical input is a string (character vector of length 1).

To aid in what could be a repetitive task, the package offeres the `concat()` function. For example:

```
> letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
    "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
> library(ngram)
> concat(letters)
[1] "abcdefghijklmnopqrstuvwxyz"
```

3.2 Package Use and Example

The general process goes

1. Prepare the input string.
2. Process with `ngram()`.
3. Generate nonesense with `babble()` and/or

3.5 Extract pieces of the processed ngram data with the `get.*()` functions.

For example, consider the string `A B A C A B B`. This is the blood code for Mortal Kombat 1, but you can pretend it's a biological sequence or something boring if you prefer. If we store this string in `x`:

```
library(ngram)
x <- "A B A C A B B"
```

then the next step is to process with `ngram()`:

```
ng <- ngram(x, n=2)
```

We can then inspect the sequence:

```
> ng
[1] "An ngram object with 5 2-grams"
```

If you don't have too many n-grams, you can print all of them by calling `print()` directly, with option `full=TRUE`:

```
> print(ng, full=TRUE)
C A
B {1} |

B A
C {1} |

B B
NULL {1} |

```

```
A C
A {1} |

A B
A {1} | B {1} |
```

Here we see each 3-gram, followed by its next possible "words" and each word's frequency of occurrence (occurrence following the given n-gram). So in the above, the first n-gram printed `C A` has `B` as a next possible word, because the sequence `C A` is only ever followed by the "word" `B` in the input string. On the other hand, `A B` is followed by `A` once and `B` once. The sequence `B B` is terminal, i.e. followed by nothing; we treat this case specially.

Next, we might want to generate some new strings. We for this, we use `babble()`:

```
> babble(ng, 10)
[1] "A C A B B B A C A B "
> babble(ng, 10)
[1] "B B C A B A C A B A "
> babble(ng, 10)
[1] "A C A B A C A B A C "
```

This generation includes a random process. For this, we developed our own implementation of MT19937, and so R's seed management does not apply. To specify your own seed, use the `seed=` argument:

```
> babble(ng, 10, seed=10)
[1] "A C A B A C A B B B "
> babble(ng, 10, seed=10)
[1] "A C A B A C A B B B "
> babble(ng, 10, seed=10)
[1] "A C A B A C A B B B "
```

3.3 Notes About the Internal Representation