Guide to the ngram Package

Fast n-gram Tokenization

Drew Schmidt and Christian Heckendorf

Guide to the **ngram** Package

Fast n-gram Tokenization

March 13, 2022

Drew Schmidt
wrathematics@gmail.com

Christian Heckendorf
heckendorfc@gmail.com

Version 3.2.1

This manual may be incorrect or out-of-date. The authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

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

This publication was typeset using LaTeX.

Contents

1 Introduction

An n-gram is an ordered sequence of n "words" taken from a body of text. For example, consider the string formed by the sequence of characters `A B A C A B B`. This is the "blood code" for the video game *Mortal Kombat* for the Sega Genesis, but you can pretend it's a biological sequence or something boring if you prefer. If we examine the 2-grams (or bigrams) of this sequence, they are:

```
A B, B A, A C, C A, A B, B B
```

or without repetition:

```
A B, B A, A C, C A, B B
```

That is, we take the input string and group the "words" 2 at a time (because `n=2`). If we form all of the n-grams and record the next "words" for each n-gram (and their frequency), then we can generate new text which has the same statistical properties as the input.

The **ngram** package (Schmidt, 2016) is an R package for constructing n-grams and generating new text as described above. It also contains a few preprocessing utilities to aid in this process. Additionally, the C code underlying this library can be compiled as a standalone shared library.

2 Installation

2.1 Installing from Source

The sourcecode for this package is available (and actively maintained) on GitHub. To install an R package from source on Windows, you will need to first install the Rtools package. To install an R package from source on a Mac, you will need to install the latest Xcode, which you can get from the App store.

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("wrathematics/ngram")
```

from R.

2.2 Installing from CRAN

The usual

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

from an R session should do it.

3 Background and Utilities

The n-gram processor in the **ngram** package changes its behavior depending on the way the input is formatted. If the input is given as a single string, n-grams crossing "sentence" boundaries will be considered valid. To prevent this from occuring, a vector of sentences may be used as input rather than a single string containing the entire text.

The n-gram tokenizer in the **ngram** package accepts a custom string containing characters to be used as word separators. There may be texts that use a wide variety of word separators, making it impractical to generate a string containing all of them.

The **ngram** package offers several useful utilities to simplify the text and assist with transforming the input to get the appropriate sentence handling behavior.

3.1 I/O

Generally speaking, the facilities in base R should be sufficient. Specifically, `readLines()` is a good choice for handling I/O of plain text files. However, for reading in multiple files (say, all `.txt` files in some directory), we offer a simple utility `multiread()`. This function will read all specified files into a named list, where the names are the filenames, and the values are the text contained in the given file as a single string.

So for example, if you want to read in all files ending in `.txt` at some directory/path of interest `path`, you could call:

```r
library(ngram)
multiread(path, extension="txt")
```

In `multiread()`, the extensions `*.txt`, `*txt`, `.txt`, and `txt` are all treated the same.

3.2 Concatenating Multiple Strings

Since the n-gram tokenizer expects only single strings, we offer the simple `concatenate()` function:

```r
> str(letters)
## chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" ...

concatenate(letters)
## [1] "a b c d e f g h i j k l m n o p q r s t u v w x y z"

concatenate(concatenate(letters, collapse=""), letters)
## [1] "abcdefghijklmnopqrstuvwxyz a b c d e f g h i j k l m n o
   p q r s t u v
w x y z"
```

So if data is coming from multiple files, the simplest way to merge them together would be to call `multiread()` (described above) if possible. Then you can call `concatenate()` directly on the returned list to produce a single string from all files, or use the tokenizer iteratively, using, say, an `lapply()`. Here is a more explicit example without the use of `multiread()`:

```
x <- readLines("file1")
y <- readLines("file2")

str <- concatenate(x, y)
```

3.3 Splitting Strings

The **ngram** tokenizer always splits words at one or more of the characters provided in the `sep` argument. You can preprocess the input string with R's regular expression utilities, such as `gsub()`. But for most tasks, the `preprocess()` and `charsplitter` utilities in the **ngram** package should be more than sufficient.

The `preprocess()` function is a simple utility for making letter case uniform, as well as optionally splitting at punctuation (so that punctuation itself becomes a "word" in the n-gram analysis). Here is a simple example:

```
x <- "Watch  out    for snakes!  "

preprocess(x)
## [1] "watch out for snakes!"

preprocess(x, case="upper", remove.punct=TRUE)
## [1] "WATCH OUT FOR SNAKES"
```

Perhaps more useful is the `charsplitter()` function. Suppose that for the purposes of n-gram tokenization, instead of wanting to call things separated by spaces "words", you wish to treat every letter as a "word". This could be accomplished by using `sep=""` when calling `ngram()`, but for the sake of introducing `charsplitter()`, it can also be done simply in this way:

```
x <- "abacabb"
splitter(x, split.char=TRUE)
## [1] "a b a c a b b"
```

By default, this will preserve spaces as a special token (an underscore by default). You may wish to ignore spaces entirely during tokenization. This too is simple to handle during preprocessing:

```
y <- "abacabb abacabb"

splitter(y, split.space=TRUE)
## [1] "abacabb _ abacabb"
splitter(y, split.space=FALSE, split.char=TRUE)
## [1] "a b a c a b b a b a c a b b"
splitter(y, split.space=TRUE, split.char=TRUE)
## [1] "a b a c a b b _ a b a c a b b"
```

3.4 Dealing with tm

The **tm** package (Feinerer et al., 2008) requires that all data be in the form of its fairly complicated `Corpus` object. The **ngram** package offers no direct methods for dealing with data in this format. To use **tm**-encapsulated data, you will first need to extract it into a single string or a vector of strings depending on what processing behavior is required.

If you want to extract the text from all documents in a corpus as a single string, you can do something like:

```r
str <- concatenate(lapply(myCorpus, "[", 1))
```

3.5 Summarizing

While not strictly related to n-gram modeling, you may wish to get some basic summary counts of your text. With the assumption that the text is a single string with words separated by one or more spaces, we can very quickly generate these counts via the `string.summary()` function:

```r
x <- "a b a c a b b"
string.summary(x)
## Chars:       13
## Letters:     7
## Whitespace:  6
## Punctuation: 0
## Digits:      0
## Words:       7
## Sentences:   0
## Lines:       1
## Wordlens:    0 7
##              9 1
## Senlens:     0
##              10
## Syllens:     0 3
##              9 1
```

Now, this "model" is based only on very simple counts, and is easily fooled. For example, the sentence `S.T.A.R. Labs is a research facility in the DC comics universe.` would be treated as though it were 5 separate sentences. However, the counts are constructed *extremely* quickly, and so they are still useful as a first pass in an analysis.

4 Using the Package

The general process for using the **ngram** package goes something like:

1. Prepare the input string; you may find the utilities in Section 3 useful.
2. Tokenize with the `ngram()` function.
3. Generate new text with `babble()`, and/or

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

4.1 Creating

Let us return to the example sequence of letters from Section 1. If we store this string in `x`:

```
x <- "A B A C A B B"
```

The next step is to process with `ngram()`:

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

Simple as that! And the tokenization was designed to be extremely fast; see Section 5 for benchmarks.

4.2 Printing

With `ng` as above, we can then inspect the sequence:

```
ng
## An ngram object with 5 2-grams
```

If you don't have too many n-grams, you may want to print all of them by calling `print()` directly, with the `print()` argument `output="full"`:

```
print(ng, output="full")
## C A | 1
## B {1} |
##
## B A | 1
## C {1} |
##
## B B | 1
## NULL {1} |
##
## A C | 1
## A {1} |
##
## A B | 2
## A {1} | B {1} |
```

Here we see each 3-gram, followed by its next possible "words" and each word's frequency of 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.

You may just wish to see the first few n-grams; this too is possible, but note that the order here is not particularly informative, in that the first n-gram shown is not necessarily the most/least common, etc. We can achieve this with the `print()` argument `output="truncated"`. However, in our example, we only have 5 n-grams, and so we will not see any difference between printing with `output="full"` versus `output="truncated"`. So we will construct a slightly more complicated example:

```
text <- rcorpus(100, alphabet=letters[1:3], maxwordlen=1)
ng2 <- ngram(text)

ng2
## An ngram object with 9 2-grams

print(ng2, output="truncated")
## b a | 14
## b {2} | a {1} | c {1} | a {2} | b {1} | a {1} | c {1} | a {1} | c {1} | b {2}
## | NULL {1} |
##
## c a | 10
## a {1} | b {1} | a {1} | c {1} | a {1} | b {1} | a {1} | b {1} | c {1} | a {1}
## |
##
## a a | 12
## c {1} | b {1} | a {1} | b {1} | c {2} | b {1} | a {1} | c {3} | b {1} |
##
## a b | 12
## b {1} | a {1} | c {1} | a {1} | b {1} | a {3} | c {2} | b {2} |
##
## c c | 4
## a {1} | b {3} |
##
## [[ ... results truncated ... ]]
```

4.3 Summarizing

Once the `ngram` representation of the text has been generated, it is very simple to get some interesting summary information. The function `get.phrasetable()` generates a “phrasetable”, or more explicitly, a table of n-grams, and their frequency and proportion in the text:

```
get.phrasetable(ng)
##   ngrams freq      prop
## 1    A B    2 0.3333333
## 2    C A    1 0.1666667
## 3   B A     1 0.1666667
## 4   B B     1 0.1666667
## 5   A C     1 0.1666667
```

We can perhaps better see the value of this in a more interesting string:

```
set.seed(12345)
text <- rcorpus(100, alphabet=letters[1:3], maxwordlen=1)
text
## [1] "a b c b b c b c a a b b c b c c c b a a b b a c c c c a
   a c a c c b c c
## c a a c b c a b a c c b c b c b c b a c c b c b c b c c b b c
   a c b c a c a b
## c c c c c a b a a c c b c a a c c a a c a a c a b"

head(get.phrasetable(ngram(text, n=3)))
##   ngrams freq       prop
## 1 c b c    12 0.12244898
## 2 b c b     8 0.08163265
## 3 c c c     7 0.07142857
## ## 4 c a a      6 0.06122449
## 5 a a c     6 0.06122449
## 6 c c b     6 0.06122449
```

Presently, there are two other "getters", namely `get.ngrams()` and `get.string()`. Each of these basically does what it sounds like. The first produces the unique n-grams as a vector of strings (in no particular order), while the second produces the input string that was used during tokenization:

```
> get.ngrams(ng)
[1] "C A" "B A" "B B" "A C" "A B"
> get.string(ng)
[1] " "
```

4.4 Babbling

We might want to use n-grams as god intended: amusement. We can easily generate new strings with the same statistical properties as the input strings via a very simple markov chain/sampling scheme. We for this, we use `babble()`:

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

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 "
```

4.5 Important Notes About the Internal Representation

The entirety of the interesting bits of the **ngram** package take place outside of R (completely in C). Observe:

```
str(ng)
## Formal class 'ngram' [package "ngram"] with 6 slots
##   ..@ str_ptr:<externalptr>
##   ..@ strlen : int 1
##   ..@ n      : int 2
##   ..@ ngl_ptr:<externalptr>
##   ..@ ngsize : int 5
##   ..@ sl_ptr :<externalptr>
```

So everything is wrangled up top as an S4 class, and underneath the data is stored as 2 linked lists, outside the purview of R. This means that, for example, that you cannot save the n-gram object with a call to `save()`. If you do and you shut down and restart R, the pointers will no longer be valid.

Extracting a the data into a native R data structure is not currently possible. Full support is planned for a later release. Some pieces can be extracted. At this time, `get.ngrams()` and `get.string()` are implemented, but `get.nextwords()` is not.

```
get.nextwords(ng)
# Error in .local(ng, ...) : Not yet implemented
```

5 Benchmarks

5.1 tau

The **tau** (Buchta et al., 2015) package offers, among other things, a framework for constructing n-grams from a text, via its `textcnt()` function.

In **ngram**, the use of `get.phrasetable(ngram(x, n=3))` roughly corresponds to `textcnt(x, n=3, split=" ", method="string")` in **tau**. Although `get.phrasetable()` returns proportions in addition to counts, and in the form of a more costly dataframe compared to **tau**'s vector of counts, we are still able to achieve very good performance.

```r
library(rbenchmark)
library(tau)
library(ngram)

x <- ngram::rcorpus(50000)

reps <- 15
cols <- c("test", "replications", "elapsed", "relative")

benchmark(tau=textcnt(x, n=3, split=" ", method="string"),
   ngram=get.phrasetable(ngram(x, n=3)), replications=reps,
   columns=cols)
```

Evaluating this script gives:

```
##    test replications elapsed relative
## 2 ngram           15   0.958    1.000
## 1   tau           15 137.994  144.044
```

In fact, a good portion of the time in the **ngram** runs here is in converting the internal C data structure over to an R one. The original purpose of the **ngram** package was merely amusement, babbling n-grams. If we just compare the run times for this, the difference is striking:

```r
library(tau)
library(ngram)

x <- ngram::rcorpus(100000)

tautime <- system.time(pt1 <- textcnt(x, n=3, split=" ",
   method="string"))[3]
ngtime <- system.time(pt2 <- ngram(x, n=3))[3]

cat("tau: ", tautime, "\n")
cat("ngram: ", ngtime, "\n")
cat("tau/ngram: ", tautime/ngtime, "\n")
```

If we evaluate this, we see:

```
## tau:  36.576
## ngram:  0.048
## tau/ngram:  762
```

Here, **ngram** is primed for babbling, in that it has already stored all “next words”, while **tau** only contains what we call the phrasetable of 3-grams.

5.2 RWeka

The **ngram** package has a separate tokenizer to produce returns similar to those in the **RWeka** package. However, **ngram** is significantly faster:

```
library(memuse)
library(ngram)
library(RWeka)

x = ngram::rcorpus(nwords=1e6, alphabet="a")
memuse(x)
## 4.292 MiB

system.time(ngram_asweka(x, min=2, max=2))
##    user  system elapsed
##   0.216   0.044   0.261

system.time(NGramTokenizer(x, Weka_control(min=2, max=2)))
##    user  system elapsed
## 500.228   0.528 500.056
```

References

Christian Buchta, Kurt Hornik, Ingo Feinerer, and David Meyer. *tau: Text Analysis Utilities*, 2015. URL http://CRAN.R-project.org/package=tau. R package version 0.0-18.

Ingo Feinerer, Kurt Hornik, and David Meyer. Text mining infrastructure in r. *Journal of Statistical Software*, 25(5):1–54, March 2008. URL http://www.jstatsoft.org/v25/i05/.

Drew Schmidt. ngram: Fast n-gram tokenization, 2016. URL https://cran.r-project.org/package=ngram. R package version 3.0.0.