Extracted from:

# Text Processing with Ruby

## Extract Value from the Data That Surrounds You



The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

# Text Processing with Ruby

## Extract Value from the Data That Surrounds You



Rob Miller

edited by Jacquelyn Carter

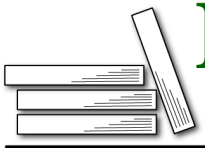
# Text Processing with Ruby

## Extract Value from the Data That Surrounds You

Rob Miller

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Cathleen Small; Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.



Printed in the United States of America.
ISBN-13: 978-1-68050-070-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 11

# Natural Language Processing

Natural language processing is an enormously broad subject, but in essence it covers any task in which we attempt to use a computer to understand human language. That might mean using a program to extract meaning from text written by a human—to translate it, to determine its subject, or to parse its grammar. Or it might mean using a computer to actually write text for humans to read—think of a computer working as a journalist, writing summaries of sports games, financial events, and so on.

## What Is Natural Language Processing?

As a field, natural language processing (or *NLP*) features many of the hardest problems that exist in computer science. A problem like machine translation, for example, involves almost every aspect of artificial intelligence: not just understanding the language syntactically, but also discerning the sentiment behind it, knowing of and deciding between the multiple and contradictory meanings that words and phrases might have, understanding all of the objects and people and places that might be referred to, and using the context of surrounding words to make judgments about potential ambiguities. These are not, for now at least, areas in which computers' greatest strengths lie.

For this reason, many solutions to problems in this area are more about distinguishing between shades of gray than they are about giving clear, black-and-white answers. But if we can get a computer to produce an answer that's good enough for our purposes, then we can exploit the things that computers *are* naturally good at—mainly that they can process text in far greater quantities and at far greater speeds than even an army of people ever could.

NLP, then, is generally a quest toward this *good enough*, which naturally varies from project to project. We're not going to venture very far into this incredibly deep subject in just one chapter; we'll just dip our toes into the

water. Even the simpler tasks we can perform with NLP are useful and worth explorating, and its complexity shouldn't dissuade you at all. After all, the thing that makes language processing so useful is that every one of us consumes—and these days creates—vast quantities of natural language text every day. It's easy to think of potential uses for natural language processing.

In our exploration of NLP, we'll look first at *text extraction* and its related problem of *term extraction*. This will allow us to extract the text of an article from its surroundings and then form an understanding about what the article's subject might be by extracting keywords from it.

Then we'll explore *fuzzy matching*. This will allow us to search through text and look not just for exact matches of a search string, as we have done previously, but matches that are slightly different—accounting not just for typing errors in the search term and the source text, but also matching variations of the same word.

In each of these examples, we'll be taking a practical look at when a technique might be useful and what ready-made libraries and tools exist to perform that technique. This isn't about implementing algorithms or exploring theory; it's about solving problems as you would in the real world—and also giving you a taste of what can be done with NLP. Let's get started.

## Example: Extracting Keywords from Articles

Let's imagine that we have a list of many hundreds of articles, spread across many different web pages. We'd like to analyze the content of these articles and create a searchable database of them for ourselves. We'd like to store the content of the articles—but not any extraneous text from the web page, such as header text or sidebar content. We'd also like to make an attempt to store some keywords, so that we can search against them and not have to search the whole body of the text. When we're finished, we'll be able to list all the terms mentioned in an article, and by extension we'll be able to list all the articles that match a particular term.

This problem neatly covers two areas of language processing. The first is *text extraction*: how we identify, given a mishmash of HTML, what the primary text on a web page is—the content of the article. The second is *term extraction*: given that text, how we extract just the key ideas and concepts that the article is about, leaving behind the useless *the*s and *he*s and other things that would fail to distinguish between this and any other article.

These two tasks are actually fairly distinct, so let's dive in and explore them both.

## Extracting Only the Article Text

Our first step in solving this problem is to download the web pages and, for each of them, extract only the text of the article. We're not interested in anything that might also be on the web page—headers, footers, and so on—but instead want only the text of the article.

As you might expect, this is largely a solved problem. There are several Ruby libraries that perform such extraction. One such library is ruby-readability,[1] a port of a tried and true library that forms the basis of the website readability.com.

Readability's algorithm, like many that tackle language processing problems, is a fuzzy one. It knows several characteristics that generally point to a particular section of a web page being the article's content: the length of the text within an element, whether there is continuous text next to the current element, the density of links within the text (to rule out things like lists of navigation), even things like the names of classes and IDs on the HTML elements themselves. Searching through the document, it assigns a score to each element based on these criteria and then extracts the best-looking matches.

While the algorithm isn't perfect, it's surprisingly capable and will definitely work for our purposes. Let's see how we can use it.

Our first step is to fetch the HTML of the web page, just as it was when we we looked at scraping content. Then we need to pass that HTML to Readability:

nlp/readability.rb

```
require "open-uri"
require "readability"
html = open("http://en.wikipedia.org/wiki/History_of_Luxembourg").read
document = Readability::Document.new(html)
```

That's actually all there is to it. We can now use the `content` method of the document object to access the HTML of the element that the Readability algorithm thinks represents the article text.

nlp/readability.rb

```
article = Nokogiri::HTML(document.content)
article.css("p").first.text
 # => "The history of Luxembourg refers to the history of the country of
 # Luxembourg and its geographical area."
```

1. https://github.com/cantino/ruby-readability

Since the returned article text is HTML, we can use Nokogiri to process it further. This example uses Nokogiri to extract only the first paragraph of the article, but it would be straightforward to return all the text (without HTML tags) or to search for text within the article that matches particular criteria.

Expanding this specific script to cope with multiple articles, by assembling an array of objects containing the article text, should be quite straightforward:

nlp/extracting-text.rb

```
require "open-uri"
require "readability"
urls = File.readlines("urls.txt").map(&:chomp)
Article = Struct.new(:title, :text, :terms)
articles = urls.map do |url|
  $stderr.puts "Processing #{url}..."
  html = open(url).read
  document = Readability::Document.new(html)
  title = document.title.sub(" - Wikipedia, the free encyclopedia", "")
  text = Nokogiri::HTML(document.content).text.strip
  Article.new(title, text)
end
```

First, we extract a list of URLs from a file. We assume that the text file contains one URL per line, and so the only processing we need to do is to strip the final newline from each of the lines. In the sample file, I'm using a list of Wikipedia articles. There's forty of them, arranged like so (here's the first six):

nlp/urls.txt

```
http://en.wikipedia.org/wiki/The_Colour_of_Magic
http://en.wikipedia.org/wiki/The_Light_Fantastic
http://en.wikipedia.org/wiki/Equal_Rites
http://en.wikipedia.org/wiki/Mort
http://en.wikipedia.org/wiki/Sourcery
http://en.wikipedia.org/wiki/Wyrd_Sisters
```

Then we define a Struct called Article for our articles, for convenience. It stores the title of the article and its text. We then loop over the URLs and generate for each of them one of these Article objects. (Since our articles are all from Wikipedia, we also strip out the generic title text that's common to them all.) At the end of the script, we're left with an array of articles that we might then store somewhere on disk or process further; in this case, we're going to be processing them further.

We've solved the first half of our problem: we've searched through our list of URLs and extracted the text of each one of them. The second half of the problem is to parse all of this text and try to figure out what the content is about.

## Extracting Terms

Now that we've been able to extract the text of the articles that we're interested in, we need to extract keywords from them to create our index. This task is called *term extraction*.

To do that, we need to know just what we mean by *keyword*. There are infinitely many definitions, naturally, but a common approach is to look at nouns that occur above a certain frequency in the text. That frequency might be in terms of the length of text—nouns that individually make up 1 percent of the words in the text, for example. Or it might be in simple and absolute terms, such as taking the nouns that occur three or more times.

Performing this task, then, involves tagging all of the *parts of speech* in the text and then removing everything that isn't a noun. Typically, a normalization step is performed, often to convert the plural form of nouns to the singular (so that *shoe* and *shoes* are regarded as the same word, for example). The number of times each noun occurs can then be counted, and finally all those nouns that don't meet the given threshold can be removed. The end result is a list of all of the nouns and proper nouns in the text, sorted by frequency, with the *strongest* keywords first.

Again, this is largely a solved problem. There are many Ruby libraries, which implement many different algorithms. One such library is Phrasie.[2]

Phrasie implements exactly this algorithm for term extraction: it pulls all of the nouns out of the text, counts them, and then omits any that occur fewer than three times (though this threshold can be adjusted).

The result is a list of nouns that appear frequently within the text. For most articles, that's a good way to judge the content and a good thing to create an index from.

Let's extend our code to also extract terms for the article text that we've fetched:

nlp/extracting-terms.rb

```
require "open-uri"
require "readability"
require "phrasie"
require "json"
require "json/add/core"
urls = File.readlines("urls.txt").map(&:chomp)
Article = Struct.new(:title, :text, :terms)
ignored_terms = ["^", "pp", "citation", "ISBN", "Retrieved", "[edit"]
```

2. https://github.com/ashleyw/phrasie/

```
articles = urls.map do |url|
  $stderr.puts "Processing #{url}..."
  html = open(url).read
  document = Readability::Document.new(html)
  title = document.title.sub(" - Wikipedia, the free encyclopedia", "")
  text = Nokogiri::HTML(document.content).text.strip
  terms = Phrasie::Extractor.new.phrases(text)
  terms = terms.reject { |term| ignored_terms.include?(term.first) }
  Article.new(title, text, terms)
end
articles.each do |article|
  puts article.title
  puts "Keywords: #{article.terms.take(5).map(&:first).join(", ")}\n\n"
end
File.open("articles.json", "w") do |json|
  json.write(JSON.generate(articles))
end
```

First, we define a list of ignored keywords. There's always going to be something that the term extractor mistakenly thinks is a noun and extracts, or that is a noun but that you don't want to include because it's common to all articles or otherwise irrelevant. In this case, the Wikipedia articles contain several citation- and footnote-related characters that we don't want to include, so we ignore them.

Then it's simply a case of passing the article text into Phrasie's extractor. It returns for us an array of arrays. Each element of the array contains the term, the number of times it occurred in the text, and the term's *strength*. The strength of a term is simply the number of words within it. Terms with multiple words are included even if they don't meet the normal threshold of occurrences, since they're considered to be stronger.

Finally, we store the results in a JSON file so that we can use this index later without having to rerun this script. We might store this information in a database or not at all; it depends on the application.

If we run this script, we'll see Phrasie's guesses for the best five keywords in each article (again, I've shown just the first six here for reasons of space):

**nlp/extracting-terms.txt**

```
The Colour of Magic
Keywords: Twoflower, Rincewind, novel, book, Magic
The Light Fantastic
Keywords: Rincewind, Twoflower, Discworld, novel, star
Equal Rites
Keywords: Esk, Simon, staff, Discworld, wizard
Mort
Keywords: Mort, Death, novel, princess, Discworld
```

```
Sourcery
Keywords: wizard, Rincewind, Coin, Ipslore, Discworld
Wyrd Sisters
Keywords: witch, king, Tomjon, Granny, King
```

Not perfect, sure, but hopefully still impressive: the simple algorithm has yielded genuinely useful keywords for all of the articles here, and there's nothing that's wrong outright. We should hopefully feel confident about building an index based on these criteria.

We've now successfully extracted the core text from a web page, ignoring all of the irrelevant content alongside it. We've then processed that text to extract the most important keywords from it, in effect building an index of the article that we could then search—saving us from having to search the full text of the article.

Next, let's look at how we can actually perform this search, and ways that we can use more language processing techniques to improve the flexibility and accuracy of that search.