# 第2章

# **文本的歧义及其清理**

在上一章中，我们为Python以及NLTK库的学习开了一个不错的头，带您初步了解了一下如何针对一些文本资料进行一些有意义的EDA。我们用非常粗糙和简单的方式将预处理部分的所有工作都做了一遍。在本章，我们将具体来讨论**断词处理**、**词干提取**、**词形还原（lemmatization）**以及**停用词移除**这些预处理步骤。这些话题将会涉及到NLTK中所有用于处理文本歧义的工具。届时，我们将会讨论现代NLP应用中会用到的所有预处理步骤，以及实现其中某些任务的不同方法，并说明我们通常该做什么、不该做什么。总而言之，我们会为您提供关于这些工具的足够信息，以便您可以自行决定在自己的应用程序中使用怎么样的预处理工具。我们希望读者在阅读完本章之后，应该掌握：

* 所有与数据歧义相关的情况，并能运用NLTK处理它们。
* 文本清理的重要性以及我们可以用NLTK实现怎么样的常见任务

## 什么是文本歧义？

事实上，要想给文本/数据歧义这个术语一个定义是相当困难的。我会将它定义成我们从原生数据中获取一段机器可读的已格式化文本之前所要做的所有预处理工作，以及所有繁复的任务。该过程应该涉及**数据再加工**（**data munging**）、**文本清理**、**特定预处理**、**断词处理**、**词干提取**或**词形还原**、以及**停用词移除**等操作。下面我们就先来看一个基本实例，解析一个csv文件：

>>>import csv

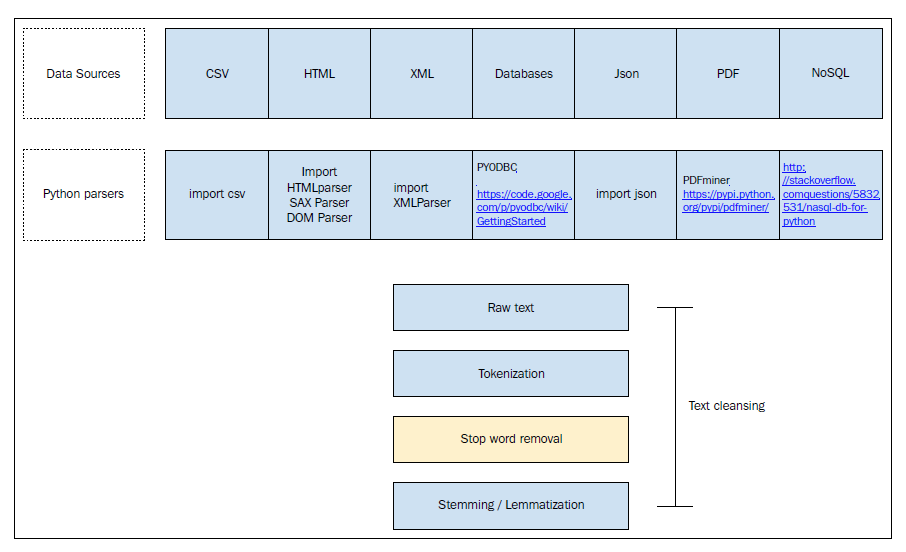
>>>with open('example.csv','rb') as f:

>>> reader = csv.reader(f,delimiter=',',quotechar='"')

>>> for line in reader :

>>> print line[1] # assuming the second field is the raw sting

如您所见，上述代码在试图对csv文件进行解析，它将会csv文件中所有的列元素构造成一个列表。我们在这操作过程中可以自定义相关的分隔符和引用符（quoting character）。现在的问题是，我们在这些原生字符串会涉及到上一章中所学到的那些不同类型的文本歧义。而其中的关键是我们要提供能应付日常csv文件的足够细节信息。



在上图中，我在堆栈的第一层中列出了一些最常见的数据源。在大多数情况下，我们将会遇到的数据都属于这些数据格式中的某一个。接下来的这一层，我列出的是Python对于这些数据格式最常见的封装方式。例如在之前那个csv文件的例子中，Python的csv模块是处理csv文件最可靠的方法。通过该模块，我们可以使用到各种不同的分离器和引用符等工具。

除此之外，json也是一种非常常见的文件格式。

下面我们来看一个具体的json实例：

{

"array": [1,2,3,4],

"boolean": True,

"object": {

"a": "b"

},

"string": "Hello World"

}

现在让我们来处理一下该字符串，其解析代码如下：

>>>import json

>>>jsonfile = open('example.json')

>>>data = json.load(jsonfile)

>>>print data['string']

"Hello World"

如您所见，我们只是用json模块加载了一个json文件。Python允许我们挑选相关原生字符串的形式并对其进行处理。关于其它所有数据源的更详细信息以及Python中相关的解析工具包，请读者自行参考我们上面列出的那个图表。当然，我们在这里只能指出相关的方向，至于这些工具包的详细信息，还需读者自己上网去搜索。

所以，在我们针对这些不同的文档格式编写自己的解析器之前，请再看一下上图第二行中所列出的Python解析器。当我们获得某一段原生字符串时，所有相关的预处理步骤都可以被用作是某一种管道，或者我们还可以选择性的忽略掉其中的部分内容。下一节，我们将会具体讨论断词处理，词干提取以及词形还原的相关细节。并且，我们也会讨论一下这些应用的各种变化，以及何时适用于其它场景。

|  |
| --- |
| 现在，既然我们对文本歧义是什么有了一点想法，就请试着用上述图表中所列出的某个Python模块连接任意一种数据库试试。 |

## 文本清理

一旦我们将各种数据源解析成了文本形式，接下来所要面临的挑战就是要使这些原生数据体现出它们的意义。文本清理就泛指这些针对文本所做的绝大部分清理，与相关数据源的依赖关系，性能的解析，外部噪声等。从这个意义上来说，这些工作和我们在*第1章：自然语言处理简介*中调用html\_clean()对HTML文档进行清理的工作是一样的。当然还有其它情况，如果我们要解析PDF文件，可能就会要清理掉一些不必要的干扰字符，移除非ASCII字符等。总之在继续下一步骤之前，我们需要做这些清理以获得一个可以被进一步处理的干净文本。而对于像XML这样的数据源，我们可能就只需要关注一些特定的树元素即可。对于数据库，我们则有各种可操作的分离器，而且有时我们也只需要关注一些特定的列。总而言之，对于所有致力于净化文本，清理掉文本周围所有可能干扰的工作，我们称之为文本清理。数据再加工（data munging）、文本清理与数据歧义这几个术语之间并没有清晰的界限，它们在类似的语境中可以相互交替使用。在接下来的几节中，我们将会具体讨论一些在任何NLP任务中都极为常见的预处理步骤。

## 语句分离器

在某些NLP应用中，我们将常常需要将一大段原生文本分割成一系列的语句，以便从中获取更多有意义的信息。直观地说，就是让语句成为一个可用的交流单元。当然，这要想在计算机上实现这个任务可比它看上去要困难得多了。典型的语句分离器既可能是（.）[[1]](#footnote-1)这样简单的字符串分割符，也有可能是某种预置分类器这样复杂的语句边界标识：

>>>inputstring = ' This is an example sent. The sentence splitter will split on sent markers. Ohh really !!'

>>>from nltk.tokenize import sent\_tokenize

>>>all\_sent = sent\_tokenize(inputstring)

>>>print all\_sent

[' This is an example sent', 'The sentence splitter will split on markers.','Ohh really !!']

在这里，我们正试着将原生文本字符串分割到一个语句列表中。用的是预处理函数sent\_tokenize()，这是一个内置在NLTK库中的语句边界检测算法。当然，如果我们在应用中需要自定义一个语句分离器的话，也可以用以下方式来调校出属于自己的语句分离器：

>>>import nltk.tokenize.punkt

>>>tokenizer = nltk.tokenize.punkt.PunktSentenceTokenizer()

该预置语句分离器可以用于17种语言。我们只需要为其指定相关的配方对象。根据我的经验，这里只要提供一个相关种类的文本语料就已经足够了，而且实际上也很少有机会需要我们自己来构建这些内容。

## 断词处理

由于在机器中，它所要理解的最小处理单位是一个单词（即*Token*）。所以除了断词处理之外，我们不宜再对这些文本字符串做更进一步的处理。所谓的断词处理，实际上就是一个将原生字符串分割成一系列有意义的token的处理过程。断词处理的复杂性因具体的NLP 应用而异，当然目标语言本身的复杂性也会带来相关的变化。例如在英语中，我们可以通过正则表达式这样简单的方式来选取纯单词内容和数字。但在中文和日文中，这会成为一个非常复杂的任务。

>>>s = "Hi Everyone ! hola gr8" # simplest tokenizer

>>>print s.split()

['Hi', 'Everyone', '!', 'hola', 'gr8']

>>>from nltk.tokenize import word\_tokenize

>>>word\_tokenize(s)

['Hi', 'Everyone', '!', 'hola', 'gr8']

>>>from nltk.tokenize import regexp\_tokenize, wordpunct\_tokenize, blankline\_tokenize

>>>regexp\_tokenize(s, pattern='\w+')

['Hi', 'Everyone', 'hola', 'gr8']

>>>regexp\_tokenize(s, pattern='\d+')

['8']

>>>wordpunct\_tokenize(s)

['Hi', ',', 'Everyone', '!!', 'hola', 'gr8']

>>>blankline\_tokenize(s)

['Hi, Everyone !! hola gr8']

在上述处理代码中，我们用到了各种断词器（tokenizers）。我们从最简单的开始：即Python字符串类型的split()方法。这是一个最基本的断词器，使用空白符来执行单词分割。当然，split()方法本身也可以被配置成一些较为复杂的断词处理过程。因此在上面的例子中，我们其实很难找出s.split()]与word\_tokenize()这两个方法之间的差异。

word\_tokenize()方法则是一个通用的，更为强大的、可面向所有类型语料库的断词处理方法。当然，word\_tokenize()是NLTK库的内置方法。如果您不能访问它，那就说明在安装NLTK数据时除了些差错。请参照*第1章：自然语言处理简介*中的内容来安装它。

通常情况下，我们有两个最常用的断词器。首先是word\_tokenize()，这是我们的默认选择，基本上能应付绝大多数的情况。另一选择是regex\_tokenize()，这是一个为用户特定需求设计的、自定义程度更高的断词器。其它断词器绝大多数都可以通过这种正则表达式断词器来。我们也可以利用某种不同的模式来构建一个非常具体的断词器。譬如在上述代码的第8行，我们也可以基于正则表达式的断词器分割出相同的字符串。您可以用\w+这个正则表达式，它会从目标字符串中分隔出所有我们所需要的单词和数字，其他语义符号也可以通过类似的分割器来进行分离，譬如对于上述代码的第10行，我们可以使用\d+这个正则表达式。这样我们就能从目标字符串中提取出纯数字内容。

现在，您能为提取大小写单词、数字和金钱符号构建专用的正则表达式断词器吗？

提示：只需参考之前正则表达式的查询模式来使用regex\_tokenize()即可。

|  |
| --- |
| 您也可以去<http://text-processing.com/demo>这个网站找一些演示项目来参考一下。 |

## 词干提取

所谓词干提取（Stemming），顾名思义就是一个修枝剪叶的过程。这是很有效的方法，通过运用一些基本规则，我们可以在修剪枝叶的过程中得到所有的token。词干提取是一种较为粗糙的规则处理过程，我们希望用它来取得相关token的各种变化。例如eat这个单词就会有像eating、eaten、eats等变化。在某些应用中，我们是没有必要区分eat和eaten之间有何不同的，所以通常会用词干提取的方式将这种语法上的变化归结为相同的词根。由此可以看出，我们之所以会用词干提取方法，就是因为它的简单，而对于更复杂的语言案例或更复杂的NLP任务的，我们就必须要改用词形还原（lemmatization）的方法了。词形还原是一种更为健全、也更有条理的方法，以便用于应对相关词根中各种语法上的变化。

下面，我们就来看一段词干提取的具体过程：

>>>from nltk.stem import PorterStemmer # import Porter stemmer

>>>from nltk.stem.lancaster import LancasterStemmer

>>>from nltk.stem.Snowball import SnowballStemmer

>>>pst = PorterStemmer() # create obj of the PorterStemmer

>>>lst = LancasterStemmer() # create obj of LancasterStemmer

>>>lst.stem("eating") eat >>>pst.stem("shopping") shop

一个拥有基本规则的词干提取器，在像移除-s/es、-ing或-ed这类事情上都可以有70％以上的精确度，而**Porter词干提取器**使用了更多的规则，自然在执行上会有很不错的精确度。

我们创建了不同的词干提取器对象，并在相关字符串上调用其stem()方法。结果如您所见，当我们用一个简单实例来查看时，它们之间并没有太大的差别，但当多种词干提取算法介入时，我们就会看到它们在精准度和性能上的差异了。关于这方面的更多细节，我们可以去看看[http://www.nltk. org/api/nltk.stem.html](http://www.nltk.org/api/nltk.stem.html)页面上的相关信息。通常情况下，我们使用的是Porter词干提取器，如果您是在英语环境中工作，这个提取器已经够用了。当然，我们还有**Snowball提取器**这一整个提取器家族，可分别用于处理荷兰语、英语、法语、德语、意大利语、葡萄牙语、罗马尼亚语、俄语等语言。特别地，我也曾经遇到过可用来处理印地文的轻量级词干提取器：<http://research.variancia.com/hindi_stemmer>。

|  |
| --- |
| 我们会建议那些希望对词干提取进行更深入研究的人去看看关于所有词干提取器的相关研究<http://en.wikipedia.org/wiki/Stemming>[[2]](#footnote-2)。  但是，大多数用户用Porter和Snowball这两种词干提取器就足以应付大量的相关用例了。在现代的NLP应用中，人们有时候会将词干提取当作是一种预处理步骤忽略掉，因此这往往取决于我们所面对的具体领域和应用。在这里，我们也希望告诉你一个事实，即如果您希望用到某些NLP标注器，譬如词性标注（POS）、NER或某种依赖性解析器中的某些部分，那么就应该避免进行词干提取操作，因为词干提取会对相关token进行修改，这有可能会导致不同的结果。  当到讨论一般标注器时，我们还会进一步对此展开讨论。 |

## 词形还原

词形还原（lemmatization）是一种更条理化的方法，它涵盖了词根所有的文法和变化形式。词形还原操作会利用上下文语境和词性来确定相关单词的变化形式，并运用不同的标准化规则，根据词性来获取相关的词根（也叫*lemma*）。

>>>from nltk.stem import WordNetLemmatizer

>>>wlem = WordNetLemmatizer()

>>>wlem.lemmatize("ate")

eat

在这里，WordNetLemmatizer使用了wordnet，它会针对某个单词去搜索wordnet这个语义字典。另外，它还用到了变形分析，以便直切词根并搜索到那个特殊的词形（即这个单词的相关变化）。因此在我们的例子中，通过*ate*这个变量是有可能会得到*eat*这个单词的，而这是词干提取操作无法做到的事情。

* 现在您能解释词干提取与词性还原之间有什么区别了吗？
* 现在您能为自己的母语设计一个Porter词干提取器（基于规则）了吗？
* 为什么像中文这样的语言来说，词干提取器是很难实现的？

## 停用词移除

停用词移除（Stop word removal）是在不同的NLP应用中最常会用到的预处理步骤之一。该步骤的思路就是想要简单地移除掉语料库中在所有文档中都会出现的单词。通常情况下，冠词和代词都会被列为停用词。这些单词在一些NPL任务（譬如说关于信息的检索和分类的任务）中是毫无意义的，这意味着这些单词通常不会产生很大的歧义。恰恰相反的是，在某些NPL应用中，停用词被移除之后所产生的影响实际上是非常小的。在大多数时候，给定语言的停用词列表都是一份通过人工制定的、跨语料库的、针对最常见单词的列表。虽然大多数语言的停用词列表都可以在相关网站上被找到，但也有一些停用词列表是基于给定语料库来自动生成的。有一种非常简单的方式就是基于相关单词在文档中出现的频率来构建一个停用词列表（即该单词在文档中出现的次数），出现在这些语料库中的单词都会被当作停用词。经过这样的充分研究，我们就会得到针对某些特定语料库的最佳停用词列表。譬如，NLTK库中就内置了涵盖22种语言的停用词列表。

下面我们来具体实现一下停用词移除的整个过程，这是一段用NLTK来处理停用词的代码。当然，您也可以在*第1章：自然语言处理简介*中一样创建一个字典，然后通过查找的方法来解决这个问题。

>>>from nltk.corpus import stopwords

>>>stoplist = stopwords.words('english') # config the language name

# NLTK supports 22 languages for removing the stop words

>>>text = "This is just a test"

>>>cleanwordlist = [word for word in text.split() if word not in stoplist]

# apart from just and test others are stopwords

['test']

在上述代码片段中，我们所做的是和第1章：自然语言处理简介中一样的停用词移除操作，但这里部属的是一个更为简洁的版本。之前，我们是基于查表法来做的。即使在目前情况下，NLTK内部所采用的是一个非常类似的方法。我们也会建议使用NLTK的停用词列表，因为这是一个更为标准化的列表，相比其它所有的实现都更为健全。而且，我们可以通过向该库的停用词构造器传递一个语言名称参数，来实现针对其它语言的类似方法。

* 在移除停用词的操作中，背后的数学运算是什么？
* 在停用词被移除之后，我们就可以执行哪些其它的NLP操作了？

## Rare word removal

This is very intuitive, as some of the words that are very unique in nature like names, brands, product names, and some of the noise characters, such as html leftouts, also need to be removed for different NLP tasks. For example, it would be really bad to use names as a predictor for a text classification problem, even if they come out as a significant predictor. We will talk about this further in subsequent chapters. We definitely don't want all these noisy tokens to be present. We also use length of the words as a criteria for removing words with very a short length or a very long length:

这是很直观的，因为一些在本质上是一样的名称，品牌，产品名称非常独特的话，有些噪音的人物，如HTML leftouts的，还需要针对不同的NLP任务被删除。例如，这将是非常不好用名称作为文本分类问题的预测，即使他们出来作为一个显著的预测。我们将在以后的章节谈论这个更多。我们绝对不希望所有这些嘈杂的令牌存在。我们也使用的话作为标准长度非常短或长很长的长度删除的话：

>>># tokens is a list of all tokens in corpus

>>>freq\_dist = nltk.FreqDist(token)

>>>rarewords = freq\_dist.keys()[-50:]

>>>after\_rare\_words = [ word for word in token not in rarewords]

We are using the FreqDist() function to get the distribution of the terms in the corpus, selecting the rarest one into a list, and then filtering our original corpus. We can also do it for individual documents, as well.

我们使用的是FreqDist（）函数获得在语料库中的术语的分配，选择所述稀有一成一个列表，然后过滤我们原来语料库。我们也可以做到这一点对单个文档，以及。

## Spell correction

It is not a necessary to use a spellchecker for all NLP applications, but some use cases require you to use a basic spellcheck. We can create a very basic spellchecker by just using a dictionary lookup. There are some enhanced string algorithms that have been developed for fuzzy string matching. One of the most commonly used is edit-distance. NLTK also provides you with a variety of metrics module that has edit\_distance.

这不是一个必要使用拼写检查所有NLP应用程序，但有些用例要求使用基本的拼写检查。我们可以只使用一个字典查找创建一个非常基本的拼写检查。有迹象表明，已经模糊字符串匹配开发的一些增强字符串算法。其中最常用的是编辑距离。 NLTK还为您提供了多种度量模块有edit\_distance。

>>>from nltk.metrics import edit\_distance

>>>edit\_distance("rain","shine")

3

We will cover this module in more detail in advanced chapters. We also have one of the most elegant codes for spellchecker from Peter Norvig, which is quite easy to understand and written in pure Python.

我们将介绍本模块中更详细地先进的章节。我们也有从彼得·诺维格，这是很容易理解和书面纯Python拼写检查最优雅的代码之一。

|  |
| --- |
| I would recommend that anyone who works with natural language processing visit the following link for spellcheck: <http://norvig.com/spell-correct.html> |

## Your turn

Here are the answers to the open-ended questions:

这里有答案的开放性问题：

* Try to connect any of the data base using pyodbc.  
  尝试连接任何使用pyodbc数据库。

<https://code.google.com/p/pyodbc/wiki/GettingStarted>

* Can you build a regex tokenizer that will only select words that are either small, capitals, numbers or money symbols?  
  你可以建立一个正则表达式标记生成器，将只选择要么是小，资金，数字或符号钱的话呢？

[\w+] selects all the words and numbers [a-z A-Z 0-9] and [\$] will match money symbol.  
[\ w+]选择所有的文字和数字[A-Z A-Z0-9]和[\$]将匹配金钱的象征。

* What's the difference between Stemming and lemmatization?  
  什么是词干和词形还原区别？

Stemming is more of a rule-based approach to get the root of the word's grammatical forms, while lemmatization also considers context and the POS of the given word, then applies rules specific to grammatical variants. Stemmers are easier to implement and the processing time is faster than lemmatizer.  
词干更多的是基于规则的方法来获得的单词的语法形式的根源，同时也词形还原考虑上下文和定单词的POS机，然后应用特定语法变规则。词干是更容易实现，并且处理时间比lemmatizer更快。

* Can you come up with a Porter stemmer (Rule-based) for your native language?  
  你能想出一个波特词干（基于规则）为您的母语？

Hint: http://tartarus.org/martin/porterstemmer/python.txt

[http://Snowball.tartarus.org/algorithms/english/stemmer.html](http://Snowball.tartarus.org/algorithm%20􀁉􀁕􀁕􀁑􀀛􀀐􀀐􀁕􀁂􀁓􀁕􀁂􀁓􀁖􀁔􀀏􀁐􀁓􀁈􀀐􀁎􀁂􀁓􀁕􀁊􀁏􀀐􀀱􀁐􀁓􀁕􀁆􀁓􀀴􀁕􀁆􀁎􀁎􀁆􀁓􀀐􀁑􀁚􀁕􀁉􀁐􀁏􀀏􀁕􀁙􀁕s/english/stemmer.html)

* Can we perform other NLP operations after stop word removal?  
  我们可以停止词删除后执行其他操作NLP？

*No*; never. All the typical NLP applications like POS tagging, chunking, and so on will need context to generate the tags for the given text. Once we remove the stop word, we lose the context.  
没有; 决不。所有典型的NLP应用，如词性标注，分块，等需要上下文来生成给定的文本标签。一旦我们删除停用词，我们失去的环境。

* Why would it be harder to implement a stemmer for languages like Hindi or Chinese?  
  为什么它会是很难实现像印地文和中国语言词干？

Indian languages are morphologically rich and it's hard to token the Chinese; there are challenges with the normalization of the symbols, so it's even harder to implement steamer. We will talk about these challenges in advanced chapters.  
印度语言形态丰富，很难令牌的中国人;有与符号正常化的挑战，因此要实现蒸笼却更难。我们将谈论先进章节这些挑战。

## 本章小结

In this chapter we talked about all the data wrangling/munging in the context of text. We went through some of the most common data sources, and how to parse them with Python packages. We talked about tokenization in depth, from a very basic string method to a custom regular expression based tokenizer.

在本章中，我们谈到了所有的数据角力/文本的情况下改写（munging）。我们通过一些最常见的数据源中去，以及如何使用Python包来解析他们。我们谈到了符号化的深入，从一个非常基本的字符串的方法来定制的基于正则表达式标记生成器。

We talked about stemming and lemmatization, and the various types of stemmers that can be used, as well as the pros and cons of each of them. We also discussed the stop word removal process, why it's important, when to remove stop words, and when it's not needed. We also briefly touched upon removing rare words and why it's important in text cleansing—both stop word and rare word removal are essentially removing outliers from the frequency distribution. We also referred to spell correction. There is no limit to what you can do with text wrangling and text cleansing. Every text corpus has new challenges, and a new kind of noise that needs to be removed. You will get to learn over time what kind of pre-processing works best for your corpus, and what can be ignored.

我们谈到了词干和词形还原，以及各类可用于词干，以及优点和他们每个人的利弊。我们还讨论了停用词删除过程中，为什么它是重要的，当删除停用词，而当不是需要它。我们还简要谈到去除生僻字以及它为什么在文本中清洗，都停字和生僻字去除重要的基本上是从频率分布去除异常值。我们也提到拼写校正。有没有限制，你可以用文本争吵和文字清洗做什么。每语料库有新的挑战，并且要除去一种新的噪声的需要。您将获得去学习一段时间什么样预处理的最适合您的主体，什么可以忽略不计。

In the next chapter will see some of the NLP related pre-processing, like POS tagging, chunking, and NER. I am leaving answers or hints for some of the open questions that we asked in the chapter.

在下一章会看到一些NLP相关的预处理，如词性标注，组块和NER。我离开的答案或提示了一些我们要求章中的开放性问题。

1. 译者注：由于原作是基于英文环境来说明的，所以本书的文本处理应该以英文标点为准。 [↑](#footnote-ref-1)
2. 译者注：相应的中文页面为：<https://zh.wikipedia.org/wiki/词干提取>。 [↑](#footnote-ref-2)