# 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

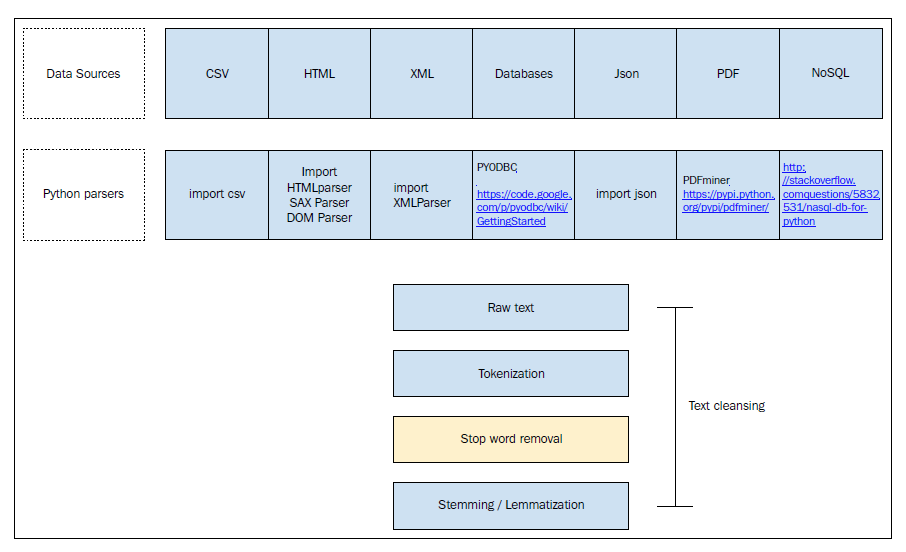
>>>with open('example.csv','rb') as f:

>>> 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。

There are two most commonly used tokenizers. The first is word\_tokenize, which is the default one, and will work in most cases. The other is regex\_tokenize, which is more of a customized tokenizer for the specific needs of the user. Most of the other tokenizers can be derived from regex tokenizers. You can also build a very specific tokenizer using a different pattern. In line 8 of the preceding code, we split the same string with the regex tokenizer. We use \w+ as a regular expression, which means we need all the words and digits from the string, and other symbols can be used as a splitter, same as what we do in line 10 where we specify \d+ as regex. The result will produce only digits from the string.

以上是我们最常用的两种断词器。第一是word\_tokenize，这是默认的，并且在大多数情况下工作。另一种是regex\_tokenize，这是更为用户的特定需求的定制标记生成的。大多数其他断词可以从正则表达式断词导出。您也可以建立一个非常具体的分词器使用不同的图案。在上面的代码第8行，我们分手的正则表达式标记生成器相同的字符串。我们用\ w +为正则表达式，这意味着我们需要所有的文字和数据从字符串，和其他符号可以被用作一个分路器，同我们在第10行做我们指定\ D +为正则表达式。结果将从字符串只生产数字。

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？

Hint: Just look for the regular expression for the preceding query and use a regex\_tokenize.

提示：只要看看对前面的查询正则表达式，并使用regex\_tokenize。

|  |
| --- |
| You can also have a look at some of the demos available online: http://text-processing.com/demo. |

## Stemming

Stemming, in literal terms, is the process of cutting down the branches of a tree to its stem. So effectively, with the use of some basic rules, any token can be cut down to its stem. Stemming is more of a crude rule-based process by which we want to club together different variations of the token. For example, the word *eat* will have variations like eating, eaten, eats, and so on. In some applications, as it does not make sense to differentiate between eat and eaten, we typically use stemming to club both grammatical variances to the root of the word. While stemming is used most of the time for its simplicity, there are cases of complex language or complex NLP tasks where it's necessary to use lemmatization instead. Lemmatization is a more robust and methodical way of combining grammatical variations to the root of a word.

词根，在字面意义，是一种树的分支削减至其干的过程。因此有效的，与使用的一些基本规则，任何标记可以削减到它的茎。词干是更粗基于规则的过程，通过它，我们要联合起 来令牌的不同变化。例如，单词吃都会有喜欢吃，吃了变化，吃，等等。在某些应用中，因为它没有任何意义区分之间吃，吃了，我们通常使用所产生的以俱乐部都语法差异字的根源。而所产生的使用的大部分时间为它的简单，有复杂的语言或复杂NLP任务的地方，有必要使用词形还原代替例。词形还原是语法变化相结合，一个字的根的更健壮和有条理的方式。

In the following snippet, we show a few stemmers:

在下面的片段中，我们展示了一些词干：

>>>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

A basic rule-based stemmer, like removing *–s/es* or *-ing* or *-ed* can give you a precision of more than 70 percent, while **Porter stemmer** also uses more rules and can achieve very good accuracies.

一个基本的基于规则的词干，如消除-s/ ES或-ing或-ed可以给你70％以上的精度，而波特词干也使用了更多的规则，可以达到很好的精度。

We are creating different stemmer objects, and applying a stem() method on the string. As you can see, there is not much of a difference when you look at a simple example, however there are many stemming algorithms around, and the precision and performance of them differ. You may want to have a look at [http://www.nltk. org/api/nltk.stem.html](http://www.nltk.org/api/nltk.stem.html) for more details. I have used Porter Stemmer most often, and if you are working with English, it's good enough. There is a family of **Snowball stemmers** that can be used for Dutch, English, French, German, Italian, Portuguese, Romanian, Russian, and so on. I also came across a light weight stemmer for Hindi on <http://research.variancia.com/hindi_stemmer>.

我们创建不同词干的物体，并在弦施加干（）方法。正如你所看到的，没有太大的差别，当你看一个简单的例子，但有很多周围而产生的算法，精度和他们的表现有所不同。你可能想看看在http：//www.nltk。组织/ API/ nltk.stem.html了解更多详情。我用波特最施特默通常，如果你用英语工作，这是不够好。有雪球词干的一个家庭，可用于荷兰语，英语，法语，德语，意大利语，葡萄牙语，罗马尼亚语，俄语，等等。我也遇到了一个重量轻词干的印地文上<http://research.variancia.com/hindi_stemmer>。

|  |
| --- |
| I would suggest a study of all the stemmers for those who want to explore more about stemmers on http://en.wikipedia.org/wiki/ Stemming.  But most users can live with Porter and Snowball stemmer for a large number of use cases. In modern NLP applications, sometimes people even ignore stemming as a pre-processing step, so it typically depends on your domain and application. I would also like to tell you the fact that if you want to use some NLP taggers, like Part of Speech tagger (POS), NER or dependency parser, you should avoid stemming, because stemming will modify the token and this can result in a different result.  We will go into this further when we talk about taggers in general. |

## Lemmatization

Lemmatization is a more methodical way of converting all the grammatical/inflected forms of the root of the word. Lemmatization uses context and part of speech to determine the inflected form of the word and applies different normalization rules for each part of speech to get the root word (*lemma*):

词形还原是将单词的根的所有语法/词尾变化的形式的更有条理的方式。词形还原使用背景和讲话的一部分，以确定所述单词的词尾变化的形式并用于语音以获得根字（引理）的每个部分施加不同的规范化规则：

>>>from nltk.stem import WordNetLemmatizer

>>>wlem = WordNetLemmatizer() >>>wlem.lemmatize("ate") eat

Here, WordNetLemmatizer is using wordnet, which takes a word and searches wordnet, a semantic dictionary. It also uses a morph analysis to cut to the root and search for the specific lemma (variation of the word). Hence, in our example it is possible to get *eat* for the given variation *ate*, which was never possible with stemming.

在这里，WordNetLemmatizer使用WordNet的，这需要一个字和WORDNET搜索，语义字典。它还采用了变形分析，切根和搜索特定引理（这个词的变化）。因此，在我们的例子中，可以得到吃的给吃了变化，这是从来没有可能的制止。

* Can you explain what the difference is between Stemming and lemmatization?
* Can you come up with a Porter stemmer (Rule-based) for your native language?
* Why would it be harder to implement a stemmer for languages like Chinese?
* 你能解释一下有什么区别词干和词形还原的？
* 你能拿出波特词干（基于规则）为您的母语？
* 为什么它会是很难实现像中国语言词干？

## Stop word removal

Stop word removal is one of the most commonly used preprocessing steps across different NLP applications. The idea is simply removing the words that occur commonly across all the documents in the corpus. Typically, articles and pronouns are generally classified as stop words. These words have no significance in some of the NLP tasks like information retrieval and classification, which means these words are not very discriminative. On the contrary, in some NLP applications stop word removal will have very little impact. Most of the time, the stop word list for the given language is a well hand-curated list of words that occur most commonly across corpuses. While the stop word lists for most languages are available online, these are also ways to automatically generate the stop word list for the given corpus. A very simple way to build a stop word list is based on word's document frequency (Number of documents the word presents), where the words present across the corpus can be treated as stop words. Enough research has been done to get the optimum list of stop words for some specific corpus. NLTK comes with a pre-built list of stop words for around 22 languages.

停用词去除是在不同的自然语言处理的应用中最常用的预处理步骤之一。这个想法是简单地删除横跨在语料库中所有的文件通常发生的话。通常情况下，文章和代词一般被列为停止的话。这些话在一些自然语言处理任务，如信息检索和分类，这意味着这些话不是很歧视没有意义。相反，在某些自然语言处理的应用程序停止字的去除将有很少的影响。大部分时间，对于给定语言的停止词列表是横跨语料库发生最常用词语的阱手策划列表。虽然大多数语言的停用词表可在网上，这些也都是方法来自动生成给定的语料停止单词列表。建立一个一站式的单词列表一个非常简单的方法是基于Word的文档频率（文件数量的字呈现），如果存在跨语料库的话可视为停用词。足够的研究已经完成，获得的停用词的最佳名单对一些特定主体。 NLTK自带停用词周边的22种语言预建的名单。

To implement the process of stop word removal, below is code that uses NLTK stop word. You can also create a dictionary on a lookup based approach like we did in *Chapter 1*, *Introduction to Natural Language Processing*.

为了实现停用词去除的过程中，下面是一个使用NLTK停止字码。您还可以创建一个字典上查找基础的方法就像我们在第1章，自然语言处理一样。

>>>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']

In the preceding code snippet, we have deployed a cleaner version of the same stop word removal we did in *Chapter 1*, *Introduction to Natural Language Processing*. Previously, we were using a lookup based approach. Even in this case, NLTK internally did a very similar approach. I would recommend using the NLTK list of stop words, because this is more of a standardized list, and this is robust when compared to any other implementation. We also have a way to use similar methods for other languages by just passing the language name as a parameter to the stop words constructor.

在前面的代码片段中，我们已经部署了同样的停用词移除，我们的确在第1章，自然语言处理更清洁的版本。以前，我们使用查找为基础的方法。即使在这种情况下，内部NLTK做了一个非常类似的方法。我会建议使用的停用词列表NLTK，因为这更多的是一种标准化的名单，当比其他任何实施这一强劲。我们也有办法通过刚好路过的语言名作为参数传递给停用词的构造函数使用其他语言的类似的方法。

* What's the math behind removing stop words?
* Can we perform other NLP operations after stop word removal?
* 背后有什么去除停用词的数学吗？
* 我们可以执行停止词删除之后其他NLP操作？

## 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。

## Summary

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)