# 第2章

# **文本的歧义及其消除**

The previous chapter was all about you getting a head start on Python as well as **NLTK**. We learned about how we can start some meaningful **EDA** with any corpus of text. We did all the pre-processing part in a very crude and simple manner. In this chapter, will go over preprocessing steps like **tokenization**, **stemming**, **lemmatization**, and **stop word** removal in more detail. We will explore all the tools in NLTK for text wrangling. We will talk about all the pre-processing steps used in modern NLP applications, the different ways to achieve some of these tasks, as well as the general do's and don'ts. The idea is to give you enough information about these tools so that you can decide what kind of pre-processing tool you need for your application. By the end of this chapter, readers should know :

前面的章节都是关于你得到Python的一个良好的开端，以及NLTK。我们了解到我们如何能够启动一些有意义的EDA与文本的语料库任何。我们做了所有的预处理部分在一个非常粗糙和简单的方式。在这一章中，将投奔预处理步骤像断词，词干，词形还原，并更详细地停止词删除。我们将探讨所有的工具NLTK文本扯皮。我们将讨论所有现代NLP应用中，不同的方法来实现某些任务的预处理步骤，以及一般该做什么和不该做什么。我们的想法是要给大家介绍的这些工具足够的信息，让你可以决定你需要什么样的前处理工具，为您的应用程序。通过本章的最后，读者应该知道：

* About all the data wrangling, and to perform it using NLTK
* What is the importance of text cleansing and what are the common tasks that can be achieved using NLTK
* 关于所有数据争论以及使用NLTK执行它
* 什么是文本清洗的重要性和什么是可以使用NLTK实现的共同任务

## What is text wrangling?

It's really hard to define the term text/data wrangling. I will define it as all the pre-processing and all the heavy lifting you do before you have a machine readable and formatted text from raw data. The process involves **data munging**, **text cleansing**, **specific preprocessing**, **tokenization**, **stemming** or **lemmatization** and **stop word removal**. Let's start with a basic example of parsing a csv file:

这真的很难定义这个词的文本/数据争吵。我将它定义为所有的前处理和你之前做繁重具有原始数据机器可读和格式化文本。这个过程涉及到的数据需要改写，文字清洗，具体的预处理，断词，词干或词形还原和停止词删除。让我们先从解析csv文件的一个基本的例子：

>>>import csv

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

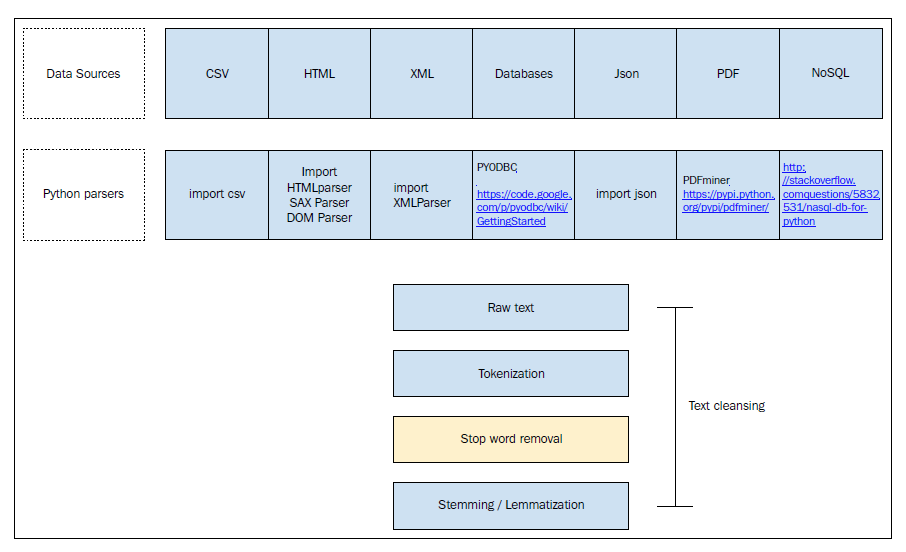
>>> reader = csv.reader(f,delimiter=',',quotechar='"')

>>> for line in reader :

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

Here we are trying to parse a csv, in above code line will be a list of all the column elements of the csv. We can customize this to work on any delimiter and quoting character. Now once we have the raw string, we can apply different kinds of text wrangling that we learned in the last chapter. The point here is to equip you with enough detail to deal with any day to day csv files.

在这里，我们试图解析CSV，在上面的代码行会csv的所有列元素的列表。我们可以自定义它的任何分隔符和引号字符工作。现在，一旦我们有原始字符串，我们可以应用不同类型的文本争吵，我们在最后一章的教训。这里的关键是使你有足够的细节，以应付任何日常的CSV文件。



I have listed most common data sources in the first stack of the diagram. In most cases, the data will be residing in one of these data formats. In the next step, I have listed the most commonly used Python wrappers around those data formats. For example, in the case of a csv file, Python's csv module is the most robust way of handling the csv file. It allows you to play with different splitters, different quote characters, and so on.

我列出最常用的数据源中的图表的第一堆栈。在大多数情况下，数据将驻留在这些数据格式之一。在接下来的步骤中，我已列出围绕这些数据格式中最常用的Python包装。例如，在一个CSV文件的情况下，Python的CSV模块是处理CSV文件中的最可靠的方法。它可以让你用不同的分离器，不同的引号字符，等玩。

The other most commonly used files are json.

其他最常用的文件是JSON。

For example, json looks like:

例如，JSON是这样的：

{

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

"boolean": True,

"object": {

"a": "b"

},

"string": "Hello World"

}

Let's say we want to process the string. The parsing code will be:

比方说，我们要处理的字符串。解析代码将是：

>>>import json

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

>>>data = json.load(jsonfile)

>>>print data['string']

"Hello World"

We are just loading a json file using the json module. Python allows you to choose and process it to a raw string form. Please have a look at the diagram to get more details about all the data sources, and their parsing packages in Python. I have only given pointers here; please feel free to search the web for more details about these packages.

我们只是加载使用JSON模块一个JSON文件。 Python允许您选择，它处理到原始字符串形式。请看看图表，以获取有关的所有数据源，并在Python语法分析程序包的更多细节。我只是这里给出的指针;请随时在网上搜索有关这些程序包的更多细节。

So before you write your own parser to parse these different document formats, please have a look at the second row for available parsers in Python. Once you reach a raw string, all the pre-processing steps can be applied as a pipeline, or you might choose to ignore some of them. We will talk about tokenization, stemmers, and lemmatizers in the next section in detail. We will also talk about the variants, and when to use one case over the other.

所以，你写你自己的解析器来解析这些不同的文件格式之前，请看看第二排在Python中使用的解析器。一旦你达到一个原始字符串，所有的预处理步骤可以应用于作为管道，或者你可以选择忽略其中的一些。我们将谈论断词，词干，并在细节下一节lemmatizers。我们也将谈论的变种，以及何时在其他使用一个案例。

|  |
| --- |
| Now that you have an idea of what text wrangling is, try to connect to any one of the databases using one of the Python modules described in the preceding image.  现在，你有什么样的文字扯皮的，尝试连接到使用前的图像中所描述的Python模块的一个数据库中的任何一个的想法。 |

## Text cleansing

Once we have parsed the text from a variety of data sources, the challenge is to make sense of this raw data. Text cleansing is loosely used for most of the cleaning to be done on text, depending on the data source, parsing performance, external noise and so on. In that sense, what we did in *Chapter 1*, *Introduction to Natural Language Processing* for cleaning the html using html\_clean, can be labeled as text cleansing. In another case, where we are parsing a PDF, there could be unwanted noisy characters, non ASCII characters to be removed, and so on. Before going on to next steps we want to remove these to get a clean text to process further. With a data source like xml, we might only be interested in some specific elements of the tree, with databases we may have to manipulate splitters, and sometimes we are only interested in specific columns. In summary, any process that is done with the aim to make the text cleaner and to remove all the noise surrounding the text can be termed as text cleansing. There are no clear boundaries between the terms data munging, text cleansing, and data wrangling they can be used interchangeably in a similar context. In the next few sections, we will talk about some of the most common preprocessing steps while doing any NLP task.

一旦我们从分析各种数据源的文本，面临的挑战是让这些原始数据的意义。文本清洗是松散用于大多数清洁的要在文本进行，这取决于数据源上，解析性能，外部噪声等。在这个意义上，我们在第一章一样，介绍自然语言处理的清洁利用html\_clean的HTML，能打成文字清洗。在另一种情况下，如果我们解析PDF，有可能是不必要的嘈杂字符，被删除非ASCII字符，依此类推。才去到下一个步骤中，我们要消除这些得到一个干净的文字进一步处理。有了这样的XML数据源，我们可能只关注在树中的某些具体内容，与数据库我们可能要操纵分离器，有时我们只是在特定的列感兴趣。总之，与该目的所做的任何方法，使文本清洁并除去周围的文字的所有的噪声可以被称为文本清洗。有数据的条件数据改写（munging），文本清洗之间没有明显的界限，并且争吵它们可以互换以类似的上下文中使用。在接下来的几节中，我们将讨论一些在做任何NLP任务中最常见的预处理步骤。

## Sentence splitter

Some of the NLP applications require splitting a large raw text into sentences to get more meaningful information out. Intuitively, a sentence is an acceptable unit of conversation. When it comes to computers, it is a harder task than it looks. A typical sentence splitter can be something as simple as splitting the string on (.), to something as complex as a predictive classifier to identify sentence boundaries:

一些NLP应用程序需要分裂一个大的原始文本为句子以获得更多的有意义的信息的。直观地说，一个句子是交谈的一个可接受的单位。当它涉及到计算机，这是一个艰巨的任务比它的外观。一个典型的句子分配器可以像分割字符串的简单的东西，东西一样复杂的预测分类，以确定句子边界（。）：

>>>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 !!']

We are trying to split the raw text string into a list of sentences. The preceding function, sent\_tokenize, internally uses a sentence boundary detection algorithm that comes pre-built into NLTK. If your application requires a custom sentence splitter, there are ways that we can train a sentence splitter of our own:

我们正试图原始文本字符串分割成句子的列表。前面的功能，sent\_tokenize，在内部使用自带预建到NLTK一个句子边界检测算法。如果应用程序需要自定义句子分配器，还有，我们可以训练我们自己的一句话分路器的方法：

>>>import nltk.tokenize.punkt

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

The preceding sentence splitter is available in all the 17 languages. You just need to specify the respective pickle object. In my experience, this is good enough to deal with a variety of the text corpus, and there is a lesser chance that you will have to build your own.

前一句分路器是所有17种语言版本。你只需要指定相应的泡菜对象。根据我的经验，这是足以应对各种语料库中，并有一个小的机会，你将不得不建立自己的。

## Tokenization

A word (*Token*) is the minimal unit that a machine can understand and process. So any text string cannot be further processed without going through tokenization. Tokenization is the process of splitting the raw string into meaningful tokens. The complexity of tokenization varies according to the need of the NLP application, and the complexity of the language itself. For example, in English it can be as simple as choosing only words and numbers through a regular expression. But for Chinese and Japanese, it will be a very complex task.

一个字（令牌）是最小单位的一台机器可以理解和处理。因此，任何文本字符串不能再无需通过符号化去处理。标记化是原始字符串分割成有意义的令牌的过程。标记化的复杂性，根据需要的NLP应用的，并且该语言本身的复杂性而变化。例如，在英语中它可以通过一个正则表达式只选择文字和数字一样简单。但中国和日本，这将是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']

In the preceding code we have used various tokenizers. To start with we used the simplest: the split() method of Python strings. This is the most basic tokenizer, that uses white space as delimiter. But the split() method itself can be configured for some more complex tokenization. In the preceding example, you will find hardly a difference between the s.split() and word\_tokenize methods.

在上面的代码中，我们使用了各种断词。首先我们用最简单的：Python字符串的split（）方法。这是最基本的标记生成器，使用白色的空间分隔符。但分裂（）方法本身可以被配置为一些更复杂的标记化。在上面的例子中，你会发现几乎没有s.split（）和word\_tokenize方法之间的差异。

The word\_tokenize method is a generic and more robust method of tokenization for any kind of text corpus. The word\_tokenize method comes pre-built with NLTK. If you are not able to access it, you made some mistakes in installing NLTK data. Please refer to *Chapter 1*, *Introduction to Natural Language Processing* for installation. 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.

该word\_tokenize方法是符号化的任何一种语料库的通用性和更强大的方法。该word\_tokenize方法带有NLTK预建。如果你不能够访问它，您在安装NLTK数据犯了一些错误。请参考第1章，自然语言处理进行安装。有两种最常用的断词。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。