# 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 :

* 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

## 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:

>>>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.

A clear process flow for some of the most commonly accepted document types is shown in the following block diagram:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Data Sources |  | CSV | HTML | XML | Databases | | Json | PDF | NoSQL | | Python parsers |  | import csv | Import  HTMLparser SAX Parser  DOM Parser | import XMLParser | | PY0DBC  https://code.google. com/p/pyodbc/wiki/  GettingStarted | import json | PDFminer  https://pypi.python. org/pypi/pdfminer/ | http:  //stackoverflow.  comquestions/5832 531/nasql-db-forpython |  |  | | --- | | Raw text |  |  | | --- | | Tokenization |  |  | | --- | | Stop word removal |  |  | | --- | | Stemming / Lemmatization |   Text cleansing |

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.

The other most commonly used files are json.

For example, json looks like:

{

"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:

>>>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.

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.

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.

## 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.

## 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:

>>>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:

>>>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.

## 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.

>>>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.

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.

Can you build a regex tokenizer that will only select words that are either small, capitals, numbers, or money symbols?

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

In the following snippet, we show a few stemmers:

>>>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.

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>.

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](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*):

>>>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.

* 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?

## 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.

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*.

>>>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.

* What's the math behind removing stop words?
* Can we perform other NLP operations after stop word removal?

## 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:

>>># 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.

## 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.

>>>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.

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:

* Try to connect any of the data base using pyodbc.

<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?

[\w+] selects all the words and numbers [a-z A-Z 0-9] and [\$] will match money symbol.

* What's the difference between Stemming and lemmatization?

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.

* Can you come up with a Porter stemmer (Rule-based) for your native language?

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?

*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.

* Why would it be harder to implement a stemmer for languages like Hindi or Chinese?

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.

## 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.

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.

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.