# Mining Twitter Data with Python Part 2: Text Pre-processing

import json

with open('mytweets.json', 'r') as f:

line = f.readline() *# read only the first tweet/line*

tweet = json.loads(line) *# load it as Python dict*

print(json.dumps(tweet, indent=4)) *# pretty-print*

The key attributes are the following:

* text: the text of the tweet itself
* created\_at: the date of creation
* favorite\_count, retweet\_count: the number of favourites and retweets
* favorited, retweeted: boolean stating whether the authenticated user (you) have favourited or retweeted this tweet
* lang: acronym for the language (e.g. “en” for english)
* id: the tweet identifier
* place, coordinates, geo: geo-location information if available
* user: the author’s full profile
* entities: list of entities like URLs, @-mentions, hashtags and symbols
* in\_reply\_to\_user\_id: user identifier if the tweet is a reply to a specific user
* in\_reply\_to\_status\_id: status identifier id the tweet is a reply to a specific status

As you can see there’s a lot of information we can play with. All the \*\_id fields also have a \*\_id\_str counterpart, where the same information is stored as a string rather than a big int (to avoid overflow problems). We can imagine how these data already allow for some interesting analysis: we can check who is most favourited/retweeted, who’s discussing with who, what are the most popular hashtags and so on. Most of the goodness we’re looking for, i.e. the content of a tweet, is anyway embedded in the text, and that’s where we’re starting our analysis.

We start our analysis by breaking the text down into words. Tokenisation is one of the most basic, yet most important, steps in text analysis. The purpose of tokenisation is to split a stream of text into smaller units called tokens, usually words or phrases. While this is a well understood problem with several out-of-the-box solutions from popular libraries, Twitter data pose some challenges because of the nature of the language.

**How to Tokenise a Tweet Text**

Let’s see an example, using the popular NLTK library to tokenise a fictitious tweet:

from nltk.tokenize import word\_tokenize

tweet = 'RT @marcobonzanini: just an example! :D http://example.com #NLP'

print(word\_tokenize(tweet))

*# ['RT', '@', 'marcobonzanini', ':', 'just', 'an', 'example', '!', ':', 'D', 'http', ':', '//example.com', '#', 'NLP']*

You will notice some peculiarities that are not captured by a general-purpose English tokeniser like the one from NLTK: @-mentions, emoticons, URLs and #hash-tags are not recognised as single tokens. The following code will propose a pre-processing chain that will consider these aspects of the language.

import re

emoticons\_str = r"""

(?:

[:=;] *# Eyes*

[oO\-]? *# Nose (optional)*

[D\)\]\(\]/\\OpP] *# Mouth*

)"""

regex\_str = [

emoticons\_str,

r'<[^>]+>', *# HTML tags*

r'(?:@[\w\_]+)', *# @-mentions*

r"(?:\#+[\w\_]+[\w\'\_\-]\*[\w\_]+)", *# hash-tags*

r'http[s]?://(?:[a-z]|[0-9]|[$-\_@.&+]|[!\*\(\),]|(?:%[0-9a-f][0-9a-f]))+', *# URLs*

r'(?:(?:\d+,?)+(?:\.?\d+)?)', *# numbers*

r"(?:[a-z][a-z'\-\_]+[a-z])", *# words with - and '*

r'(?:[\w\_]+)', *# other words*

r'(?:\S)' *# anything else*

]

tokens\_re = re.compile(r'('+'|'.join(regex\_str)+')', re.VERBOSE | re.IGNORECASE)

emoticon\_re = re.compile(r'^'+emoticons\_str+'$', re.VERBOSE | re.IGNORECASE)

def tokenize(s):

return tokens\_re.findall(s)

def preprocess(s, lowercase=False):

tokens = tokenize(s)

if lowercase:

tokens = [token if emoticon\_re.search(token) else token.lower() for token in tokens]

return tokens

tweet = "RT @marcobonzanini: just an example! :D http://example.com #NLP"

print(preprocess(tweet))

*# ['RT', '@marcobonzanini', ':', 'just', 'an', 'example', '!', ':D', 'http://example.com', '#NLP']*

As you can see, @-mentions, emoticons, URLs and #hash-tags are now preserved as individual tokens.

If we want to process all our tweets, previously saved on file:

with open('mytweets.json', 'r') as f:

for line in f:

tweet = json.loads(line)

tokens = preprocess(tweet['text'])

do\_something\_else(tokens)

The tokeniser is probably far from perfect, but it gives you the general idea. The tokenisation is based on regular expressions (regexp), which is a common choice for this type of problem. Some particular types of tokens (e.g. phone numbers or chemical names) will not be captured, and will be probably broken into several tokens. To overcome this problem, as well as to improve the richness of your pre-processing pipeline, you can improve the regular expressions, or even employ more sophisticated techniques like Named Entity Recognition.

The core component of the tokeniser is the regex\_str variable, which is a list of possible patterns. In particular, we try to capture some emoticons, HTML tags, Twitter @usernames (@-mentions), Twitter #hashtags, URLs, numbers, words with and without dashes and apostrophes, and finally “anything else”. Please take a moment to observe the regexp for capturing numbers: why don’t we just use \d+? The problem here is that numbers can appear in several different ways, e.g. 1000 can also be written as 1,000 or 1,000.00 — and we can get into more complications in a multi-lingual environment where commas and dots are inverted: “one thousand” can be written as 1.000 or 1.000,00 in many non-anglophone countries. The task of identifying numeric tokens correctly just gives you a glimpse of how difficult tokenisation can be.

The regular expressions are compiled with the flags re.VERBOSE, to allow spaces in the regexp to be ignored (see the multi-line emoticons regexp), and re.IGNORECASE to catch both upper and lowercases. The tokenize()function simply catches all the tokens in a string and returns them as a list. This function is used within preprocess(), which is used as a pre-processing chain: in this case we simply add a lowercasing feature for all the tokens that are not emoticons