Lab 2: Regular Expression

**References:**

1. Regular Expressions: The Complete Tutorial, by Jan Goyvaerts, 2007.
2. Speech and Language Processing, by Dan Jurafsky and James H. Martin. Prentice Hall Series in Artificial Intelligence, 2008.
3. Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper, 2014.

Quick Review

A regular expression (regex or regexp for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as \*.txt to find all text files in a file manager. The regex equivalent is «.\*\.txt». Basically, a regular expression is a pattern describing a certain amount of text.

A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

Practices

* 1. Extracting Word Pieces

The re.findall() ("find all") method finds all (non-overlapping) matches of the given regular expression. Let's find all the vowels in a word, then count them.

import re

word = 'supercalifragilisticexpialidocious'

print(re.findall(r'[aeiou]', word))

print(len(re.findall(r'[aeiou]', word)))

# Search

str = 'an example word:cat!!'

match = re.search(r'word:\w\w\w', str)

# If-statement after search() tests if it succeeded

if match:

print('found'), match.group() ## 'found word:cat'

else:

print('did not find')

Let's look for all sequences of two or more vowels in some text, and determine their relative frequency:

wsj = sorted(set(nltk.corpus.treebank.words()))

fd = nltk.FreqDist(vs for word in wsj

... for vs in re.findall(r'[aeiou]{2,}', word))

* 1. More Practice with Word Pieces

Once we can use re.findall() to extract material from words, there's interesting things to do with the pieces, like glue them back together or plot them.

It is sometimes noted that English text is highly redundant, and it is still easy to read when word-internal vowels are left out. For example, declaration becomes dclrtn, and inalienable becomes inlnble, retaining any initial or final vowel sequences. The regular expression in our next example matches initial vowel sequences, final vowel sequences, and all consonants; everything else is ignored. This three-way disjunction is processed left-to-right, if one of the three parts matches the word, any later parts of the regular expression are ignored. We use re.findall() to extract all the matching pieces, and ''.join() to join them together.

import re

import nltk

word = 'supercalifragilisticexpialidocious'

regexp = r'^[AEIOUaeiou]+|[AEIOUaeiou]+$|[^AEIOUaeiou]'

def compress(word):

pieces = re.findall(regexp, word)

return ''.join(pieces)

english\_udhr = nltk.corpus.udhr.words('English-Latin1')

print(nltk.tokenwrap(compress(w) for w in english\_udhr[:75]))

🗫 How to fetch internet resources (https://docs.python.org/3/library/urllib.request.html?highlight=urllib)

import urllib.request

url = "http://www.gutenberg.org/ebooks/2554?msg=welcome\_stranger"

with urllib.request.urlopen(url) as f:

print(f.read(300).decode('utf-8'))

Next, let's combine regular expressions with conditional frequency distributions. Here we will extract all consonant-vowel sequences from the words of Rotokas, such as *ka* and *si*. Since each of these is a pair, it can be used to initialize a conditional frequency distribution. We then tabulate the frequency of each pair:

import nltk, re

rotokas\_words = nltk.corpus.toolbox.words('rotokas.dic')

cvs = [cv for w in rotokas\_words for cv in re.findall(r'[ptksvr][aeiou]', w)]

cfd = nltk.ConditionalFreqDist(cvs)

print(cfd.tabulate())

Examining the rows for *s* and *t*, we see they are in partial "complementary distribution", which is evidence that they are not distinct phonemes in the language. Thus, we could conceivably drop s from the Rotokas alphabet and simply have a pronunciation rule that the letter *t* is pronounced *s* when followed by *i*. (Note that the single entry having *su*, namely *kasuari*, 'cassowary' is borrowed from English.)

If we want to be able to inspect the words behind the numbers in the above table, it would be helpful to have an index, allowing us to quickly find the list of words that contains a given consonant-vowel pair, e.g. cv\_index['su'] should give us all words containing *su*. Here's how we can do this:

import nltk, re

rotokas\_words = nltk.corpus.toolbox.words('rotokas.dic')

cv\_word\_pairs = [(cv, w) for w in rotokas\_words

for cv in re.findall(r'[ptksvr][aeiou]', w)]

cv\_index = nltk.Index(cv\_word\_pairs)

print(cv\_index['su'])

print(cv\_index['po'])

This program processes each word w in turn, and for each one, finds every substring that matches the regular expression «[ptksvr][aeiou]». In the case of the word *kasuari*, it finds *ka*, *su* and *ri*. Therefore, the cv\_word\_pairs list will contain ('ka', 'kasuari'), ('su', 'kasuari') and ('ri', 'kasuari'). One further step, using nltk.Index(), converts this into a useful index.

* 1. Finding Word Stems

When we use a web search engine, we usually don't mind (or even notice) if the words in the document differ from our search terms in having different endings. A query for laptops finds documents containing laptop and vice versa. Indeed, laptop and laptops are just two forms of the same dictionary word (or lemma). For some language processing tasks we want to ignore word endings, and just deal with word stems.

There are various ways we can pull out the stem of a word. Here's a simple-minded approach which just strips off anything that looks like a suffix:

def stem(word):

for suffix in ['ing', 'ly', 'ed', 'ious', 'ies', 'ive', 'es', 's', 'ment']:

if word.endswith(suffix):

return word[:-len(suffix)]

return word

print(stem('usually'))

* 1. Using Regular Expression for Stemming

Although we will ultimately use NLTK's built-in stemmers, it's interesting to see how we can use regular expressions for this task. Our first step is to build up a disjunction of all the suffixes. We need to enclose it in parentheses in order to limit the scope of the disjunction:

import re

print(re.findall(r'^.\*(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing'))

Here, re.findall() just gave us the suffix even though the regular expression matched the entire word. This is because the parentheses have a second function, to select substrings to be extracted. If we want to use the parentheses to specify the scope of the disjunction, but not to select the material to be output, we have to add ?:, which is just one of many arcane subtleties of regular expressions. Here's the revised version.

import re

print(re.findall(r'^.\*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing'))

However, we'd actually like to split the word into stem and suffix. So we should just parenthesize both parts of the regular expression:

import re

print(re.findall(r'^(.\*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing'))

We can change it for the user’s input:

import re

inputWord = input("Enter the word for stemming: ")

print(re.findall(r'^(.\*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', inputWord))

Test it for “processes”.

**Question:** What is the problem?

**Answer:** The regular expression incorrectly found an *-s* suffix instead of an *-es* suffix. This demonstrates another subtlety: the star operator is "greedy" and the .\* part of the expression tries to consume as much of the input as possible.

If we use the "non-greedy" version of the star operator, written \*?, we get what we want:

re.findall(r'^(.\*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', inputWord)

Test it for “language”.

**Question:** What is the problem?

Solve the problem using the following code:

re.findall(r'^(.\*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$', inputWord)

This approach still has many problems (can you spot them?) but we will move on to define a function to perform stemming, and apply it to a whole text:

import re #Python2

from nltk.tokenize import sent\_tokenize, word\_tokenize

def stem(word):

regexp = r'^(.\*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'

stem, suffix = re.findall(regexp, word)[0]

return stem

sampleText = """You probably worked out that a backslash means that

the following character is deprived of its special powers

and must literally match a specific character in the word.

Thus, while . is special, \. only matches a period. The

braced expressions, like {3,5}, specify the number of repeats

of the previous item. The pipe character indicates a choice

between the material on its left or its right. Parentheses

indicate the scope of an operator: they can be used together

with the pipe (or disjunction) symbol like this: «w(i|e|ai|oo)t»,

matching wit, wet, wait, and woot. It is instructive to see what

happens when you omit the parentheses from the last expression

above."""

tokens = word\_tokenize(sampleText)

print([stem(t) for t in tokens])

Notice that our regular expression removed the *s* from *ponds* but also from *is* and *basis*. It produced some non-words like *distribut* and *deriv*, but these are acceptable stems in some applications.

import re #Python3

import nltk

def stem(word):

regexp = r'^(.\*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'

stem, suffix = re.findall(regexp, word)[0]

return stem

sampleText = """You probably worked out that a backslash means that

the following character is deprived of its special powers

and must literally match a specific character in the word.

Thus, while . is special, \. only matches a period. The

braced expressions, like {3,5}, specify the number of repeats

of the previous item. The pipe character indicates a choice

between the material on its left or its right. Parentheses

indicate the scope of an operator: they can be used together

with the pipe (or disjunction) symbol like this: «w(i|e|ai|oo)t»,

matching wit, wet, wait, and woot. It is instructive to see what

happens when you omit the parentheses from the last expression

above."""

tokens = nltk.tokenize.word\_tokenize(sampleText)

print([stem(t) for t in tokens])

from nltk.stem import RegexpStemmer

* 1. Using Regular Expression package to perform split

Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

import re

text = "I am a human"

tokens = re.split(' ', text)

print(tokens)

# An example of split by consecutive numbers is as follows.

s\_nums = 'one1two22three333four'

print(re.split('\d+', s\_nums))

print(re.split('\d+', s\_nums, 2))

s\_marks = 'one-two+three#four'

print(re.split('[-+#]', s\_marks))

* 1. Using Python String split() method to perform split

The split() method splits a string into a list.

You can specify the separator, default separator is any whitespace.

*Note:* When maxsplit is specified, the list will contain the specified number of elements plus one.

string.split(separator, maxsplit)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| separator | Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator |
| maxsplit | Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences" |

***Example 1:*** Split the string, using comma, followed by a space, as a separator:

txt = "hello, my name is Peter, I am 26 years old"

x = txt.split(",")

print(x)

***Example 2:*** *Use a hash character as a separator:*

txt = "apple#banana#cherry#orange"

x = txt.split("#")

print(x)

***Example 3:*** *Split the string into a list with max 2 items:*

txt = "apple#banana#cherry#orange"

# setting the maxsplit parameter to 1, will return a list with 2 elements!

x = txt.split("#", 1)

print(x)

* 1. Stemming

**Stemming** with **Python** nltk package. "**Stemming** is the process of reducing inflection in words to their root forms such as mapping a group of words to the same **stem** even if the **stem** itself is not a valid word in the Language.

**Using nltk.PorterStemmer()**

In linguistic morphology and information retrieval, stemming is the process of reducing inflected words to their word stem, base or root form—generally a written word form

import nltk

Text = input("Enter the text corpus for stemming: ")

word = Text.lower().split(' ')

print(word)

porter = nltk.PorterStemmer()

for t in word:

print(porter.stem(t))

import nltk

Text = input("Enter the text corpus for stemming: ")

word = nltk.tokenize.word\_tokenize(Text)

print(word)

porter = nltk.PorterStemmer()

for t in word:

    print(porter.stem(t))