正则表达式、文本正则化、编辑距离

文本正则化的目的是将文本转换成一种更方便使用、更标准的表达形式。正则表达是一个其中的一个强有力的工具。对于大部分语言的处理，通常第一步需要做分词，这一类任务叫做**Tokenization**。另一个很重要的步骤是 **Lemmatization（词形还原，例如英文中 is, are, am 都是 be，对于中文这一步，主要是简繁转换等，主要用于处理词法复杂的语言）**。**Stemming（词干提取，通常是是分离后缀）**。文本正则化通常也包含**句子分割**，例如以句号或者感叹号分割。

编辑距离是基于编辑的次数（增删改）比较给定两个字符串之间的相似度。

2.1 Regular Expressions

类似于 Unix 下的 grep 或 Emacs。

要习惯使用在线正则表达式测试自己的表达式写的是否正确。https://regex101.com/

2.1.1 基本的正则表达式模式

- **[]**中括号内的内容是并且关系，如 [abc] 代表 "a", "b" or "c"
- **-** 短线（减号）代表区间，如 [a-z] 代表 26 个小写字母
- **^** 插入符代表否定的意思，如 [^A-Z] 代表非大写的26个字母
  - 该符号与中括号一起用表示非的意思
  - 也可以表示一个 Anochors，表示以什么开头的意思，与此对应的是 "$" 表示以什么结尾的意思
  - 也可以仅仅表示这个符号本身
- **?** 问号 表示有还是没有，0次或1次，如 [colou?r] 表示 "colour" or "color"
- ***** 星号表示任意次（包括0）。如 a* 表示空或任意长的连续个 a
- **+** 加号表示至少一次（不含0）。如 a+ 表示至少长度为 1 的连续个 a
- **{}** 花括号，表示出现多少次，如 a{3}b 表示 “aaab”
  - {n,m} n到m 次
  - {n,}
  - {,m}
- **.** 点号，通配符，匹配任意字符（除了回车符）
- 除了 **^** 和 **$**，还有两个 anchors
  - \b：匹配单词边界，例如 \bthe\b 只能匹配单词"the"，不能是 "other"，这与怎么定义单词边界有关。

2.1.2 Disjunction（析取）、Grouping、以及优先级顺序

- | 竖线，析取表达式，表示或的意思，例如 “cat|dog” 表示 “cat” or "dog"
- () 小括号，将需要优先处理的部分括起来，例如 "gupp(y|ies)" 表示 “guppy” or “guppies”
- 优先级顺序：
  - 第一级：圆括号
  - 计数的：* + ? {}
  - 序列和锚点: the ^my end$
  - 析取符号：|
- 默认的是贪婪模式，尽可能匹配符合条件的最长字符串，若想使用非贪婪模式，可以在 * 或 + 后面加一个 ？，则会尽可能短的进行匹配。

2.1.5 更多操作符（小结）

- \d： 0-9 ; \D 非0-9
- \w: 字母数字下划线； \W 非
- \s: 空白符，含[" ", \r\t\n\f]；\S 非

2.1.6 正则表达式替换、捕获组

- s/regexp1/pattern：s/colour/color 在 vim 或 sed 下可以将 colour 替换为 color
- 数字注册：\1 操作，将匹配的内容用于处理操作中，如将 "35 boxes" 替换为 "<35> boxes",
  - s/([0-9]+)/<\1>
  - 两个的示例：/the (.*)er they (.*), the \1er we \2/
- 非捕获组模式：
  - /(?:some|a few) (people|cats) like some \1/： 如果前面匹配的是 some cats, \1 的位置也必须是 some cats ， 这里不存在替换操作

Python 正则表达式可以参照这里，也可以通过 "help(re)" 的方式查看完整的 API 文档。

- re.match与 re.search 的区别：re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

2.2 Words

- **Types** 指的是词典表 V 的大小
- **Tokens** 指的是总词数 N （含有重复）

2.4 文本正则化

- Segmenting/tokenizing words from running text
- Normalizing word formats
- Segmenting sentences in running text.

2.4.1 利用 Unix 工具粗糙的分词和正则化

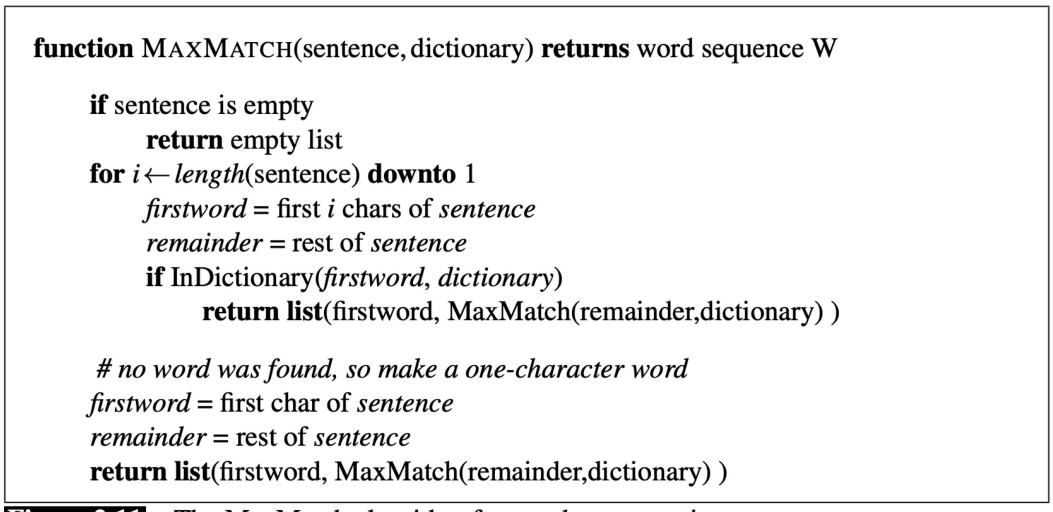
- tr 命令：可以对来自标准输入的字符串进行替换、压缩和删除。
- sort 命令：将输入行按字典顺序排序
- uniq 命令：用于报告或忽略文件中的重复行，一般与 sort 结合使用

2.4.3 中文分词：最大匹配算法

这是一个基线方法，用于对比其他更先进的算法。该算法需要提供一个分词词典。

基本思想是从[start, end] 看看在不在词典中，不在则 [start, end-1] 直到剩 start 那就单独分离一个字

整个算法伪代码如下：



function MAXMATCH(sentence, dictionary) **returns** word sequence W

if sentence is empty
return empty list
for *i* ← *length*(sentence) **downto** 1
firstword = first *i* chars of *sentence*
remainder = rest of *sentence*
if InDictionary(*firstword*, *dictionary*)
return list(firstword, MaxMatch(remainder,dictionary))

 # no word was found, so make a one-character word
firstword = first char of *sentence*
remainder = rest of *sentence*
return list(firstword, MaxMatch(remainder,dictionary))

Figure 2.11 The MaxMatch algorithm for word segmentation.

```python
# http://lion137.blogspot.com/2017/01/text-segmentation-maximum-matching-in.html
D = ['danny', 'condo', 'a', 'the', 'to', 'has', 'been', 'unable', 'go', 'at']


def max_match(sentence, dictionary):
    if not sentence:
        return ""
    for i in range(len(sentence), -1, -1):
        first_word = sentence[:i]
        remainder = sentence[i:]
        if first_word in dictionary:
```

```
            return first_word + " " + max_match(remainder,
dictionary)
    first_word = sentence[0]
    remainder = sentence[1:]
    return first_word + max_match(remainder, dictionary)


print(max_match('atcondogo', D))
```

一般用 word error rate 评估分词的质量，该算法的一个问题是无法解决 unknown words，中文分词通常使用统计序列模型。

2.4.4 词形还原以及词干提取

词形还原是用来检测两个单词是否具有相同的根，单词一般分为词素和词根两部分，词素代表该单词的主要意思，词根是额外的信息。

词形还原算法是比较复杂的，但我们可以整一个比较简单的粗糙的，比如直接去掉词缀，这种提取词干的方式较 **stemming**。最常用的 stemming 算法是 **Porter stemmer**，该算法是基于一些简单的规则进行处理的，例如，去掉以 “ing” 结尾，将 “sses” 替换成 "ss" 等。算法细节以及各种代码实现可以参照这里。

下面举一个 NLTK 工具包中的例子：

```
import nltk
from nltk.stem.porter import *

stemmer = PorterStemmer()
tokens = ['compute', 'computer', 'computed', 'computing']
for token in tokens:
    print(token + ' --> ' + stemmer.stem(token))


"""
compute --> comput
computer --> comput
computed --> comput
computing --> comput
"""
```

稍微高级一点的版本叫 "Snowball Stemmer",

```
from nltk.stem.snowball import SnowballStemmer

stemmer = SnowballStemmer(language="english")
tokens = ['compute', 'computer', 'computed', 'computing']
```

```
5
6  for token in tokens:
7      print(token + " --> " + stemmer.stem(token))
8
9  """
10 compute --> comput
11 computer --> comput
12 computed --> comput
13 computing --> comput
14 """
```

Stem 处理后的单词可能是非字典中的单词，所以是时候使用 **Lemmatization** 这个大招了。

我们借助 SpaCy 这个强大的工具包，借助 **lemma_** 属性即可。

```
1  import spacy
2  nlp = spacy.load("en_core_web_sm")
3  sentence = nlp("compute computer computed computing")
4
5  for word in sentence:
6      print(word.text, word.lemma_)
7
8  """
9  compute compute
10 computer computer
11 computed compute
12 computing computing
13 """
```

2.4.5 Byte-Pair Encoding（BPE）

Stem 和 Lemma 有一个附加的好处是，可以在一定程度上解决 **unknown words**。例如训练集中可能含有 low 和 lowest，不包含 lower，lower 出现在测试集中。但这么做可能会损失一些重要信息，例如对于词性标注，我们需要保持这种区别。

为了解决这个问题，我们采用另一种分词手法，保留大部分单词，以及诸如 "-er" 等内容，这样不认识的单词可以通过零碎的部件拼接。

```
1  import re
2  import collections
3
4
5  def get_stats(voc):
6      dict_pairs = collections.defaultdict(int)
7      for word, freq in voc.items():
8          symbols = word.split()
```

```
        for index in range(len(symbols) - 1):
            dict_pairs[symbols[index], symbols[index+1]] += freq
    return dict_pairs


def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub("".join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out


vocab = {'l o w </w>': 5, 'l o w e s t </w>': 2,
         'n e w e r </w>': 6, 'w i d e r </w>': 3, 'n e w </w>': 2}

num_merges = 8
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(best)

```

2.5 最小编辑距离

计算两个单词之间，通过增、改、插 等手段的最小变换次数。例如：

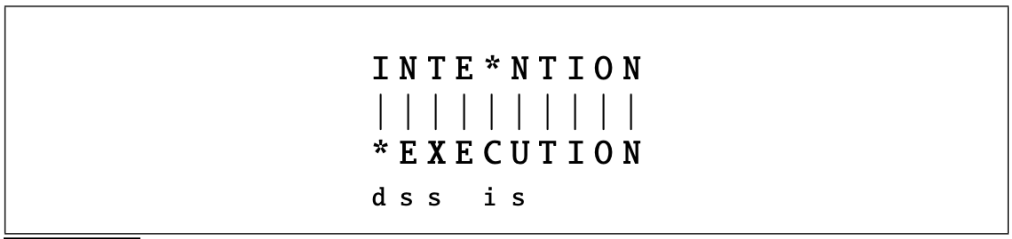


Figure 2.13 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

上面两个单词的编辑距离为 5，有的操作中不存在“改”，因为一个“改”可以由一个“删”和一个“插”组合而成，这样算则两个单词的编辑距离为 8.

最小编辑距离算法

寻找编辑距离的过程可以视为一个查找问题。所有的编辑可能空间是巨大的，所以我们不能暴力查找，这种问题的求解通常是采用**动态规划算法**

```python
import numpy as np


def normal_leven(str1, str2):
    len_str1 = len(str1) + 1
    len_str2 = len(str2) + 1

    matrix = np.zeros((len_str1, len_str2))
    matrix[0, :] = range(len_str2)
    matrix[:, 0] = range(len_str1)

    for i in range(1, len_str1):
        for j in range(1, len_str2):
            if str1[i-1] == str2[j-1]:
                cost = 0
            else:
                cost = 2
            matrix[i, j] = min(matrix[i-1, j] + 1, matrix[i, j-1] + 1, matrix[i-1, j-1] + cost)

    return matrix[-1, -1]


print(normal_leven("intention", "execution"))
```

参考：

- Speech and Language Processing (3rd ed. draft) 第二章内容
- Python for NLP: Tokenization, Stemming, and Lemmatization with SpaCy Library