L5-Python-and-Text-Processing

March 26, 2018

1 Concepts in text processing (文本处理基本概念)

1.1 Corpora (语料库)

Corpus is a large collection of texts. It is a body of written or spoken material upon which a linguistic analysis is based. A corpus provides grammarians, lexicographers, and other interested parties with better discriptions of a language. Computer-procesable corpora allow linguists to adopt the principle of total accountability, retrieving all the occurrences of a particular word or structure for inspection or randomly selcted samples. Corpus analysis provide lexical information, morphosyntactic information, semantic information and pragmatic information.

1.2 Tokens

A token is the technical name for a sequence of characters, that we want to treat as a group. The vocabulary of a text is just the set of tokens that it uses, since in a set, all duplicates are collapsed together. In Python we can obtain the vocabulary items with the command: `set()`.

1.3 Stopwords （停词）

Stopwords are common words that generally do not contribute to the meaning of a sentence, at least for the purposes of information retrieval and natural language processing. These are words such as the and a. Most search engines will filter out stopwords from search queries and documents in order to save space in their index.

1.4 Stemming （词根检索）

Stemming is a technique to remove affixes from a word, ending up with the stem. For example, the stem of cooking is cook , and a good stemming algorithm knows that the ing suffix can be removed. Stemming is most commonly used by search engines for indexing words. Instead of storing all forms of a word, a search engine can store only the stems, greatly reducing the size of index while increasing retrieval accuracy.

1.5 Frequency Counts （频数统计）

Frequency Counts the number of hits. Frequency counts require finding all the occurences of a particular feature in the corpus. So it is implicit in concordancing. Software is used for this purpose. Frequency counts can be explained statistically.

1.6 Word Segmenter （分词）

Word segmentation is the problem of dividing a string of written language into its component words.

In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter). (Some examples where the space character alone may not be sufficient include contractions like can't for can not.)

However the equivalent to this character is not found in all written scripts, and without it word segmentation is a difficult problem. Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited.

1.7 Part-Of-Speech Tagger （词性标注工具）

In corpus linguistics, part-of-speech tagging (POS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition, as well as its context—i.e. relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

1.8 Named Entity Recognizer（命名实体识别工具）

Named-entity recognition (NER) (also known as entity identification, entity chunking and entity extraction) is a subtask of information extraction that seeks to locate and classify elements in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages.

1.9 Parser（句法分析器）

Here we will treat text as `raw data` for the programs we write, programs that manipulate and analyze it in a variety of interesting ways.

2 Natural Language Processing tools （自然语言处理工具）

- Natural Language Toolkit

  NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

  Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more.

- Stanford Word Segmenter

Tokenization of raw text is a standard pre-processing step for many NLP tasks. For English, tokenization usually involves punctuation splitting and separation of some affixes like possessives. Other languages require more extensive token pre-processing, which is usually called segmentation.

The Stanford Word Segmenter currently supports Arabic and Chinese. The provided segmentation schemes have been found to work well for a variety of applications.

- NLP toolkits for Chinese
  - Toolkit for Chinese natural language processing
  - The ICT Natural Language Processing Research Group
  - Jieba Chinse Word Segmenter

3 Tokenizing Text （标记文本）

Before you use the NLTK library, please use the NLTK Downloader to obtain the resource:
`import nltk` `nltk.download('punkt')` `nltk.download('stopwords')` `nltk.download('treebank')`

The NLTK data will be downloaded in to the following director depending your OS.
`$HOME/nltk_data` `/usr/share/nltk_data` `/usr/local/share/nltk_data` `/usr/lib/nltk_data` `/usr/local/lib/nltk_data` `/usr/nltk_data` `/usr/lib/nltk_data`

3.1 Tokenizing text into sentences （断句）

The `sent_tokenize()` function uses an instance of `PunktSentenceTokenizer` from the `nltk.tokenize.punkt` module. This instance has already been trained and works well for many European languages. So it knows what punctuation and characters mark the end of a sentence and the beginning of a new sentence.

```
In [6]: para = "Python is a widely used general-purpose, high-level programming language. \
         Its design philosophy emphasizes code readability, and its syntax allows progra
         to express concepts in fewer lines of code than would be possible in languages
         C++ or Java. The language provides constructs intended to enable clear programs
         on both a small and large scale."
```

```
In [13]: from nltk.tokenize import sent_tokenize
         sent_tokenize(para)
```

```
Out[13]: ['Python is a widely used general-purpose, high-level programming language.',
          'Its design philosophy emphasizes code readability, and its syntax allows programmers
          'The language provides constructs intended to enable clear programs         on both a
```

3.2 Tokenizing sentences into words （分词）

```
In [14]: from nltk.tokenize import word_tokenize
         word_tokenize('Hello World.')
```

```
Out[14]: ['Hello', 'World', '.']
```

```
In [9]: word_tokenize(para)
```

```
Out[9]: ['Python',
         'is',
         'a',
         'widely',
         'used',
         'general-purpose',
         ',',
         'high-level',
         'programming',
         'language',
         '.',
         'Its',
         'design',
         'philosophy',
         'emphasizes',
         'code',
         'readability',
         ',',
         'and',
         'its',
         'syntax',
         'allows',
         'programmers',
         'to',
         'express',
         'concepts',
         'in',
         'fewer',
         'lines',
         'of',
         'code',
         'than',
         'would',
         'be',
         'possible',
         'in',
         'languages',
         'such',
         'as',
         'C++',
         'or',
         'Java',
         '.',
         'The',
         'language',
         'provides',
         'constructs',
         'intended',
         'to',
         'enable',
         'clear',
         'programs',
         'on',
         'both',
         'a',
         'small',
         'and',
         'large',
         'scale',
         '.']
```

3.3 Tokenizing sentences using regular expressions （使用正则表达式分词）

```
In [15]: from nltk.tokenize import RegexpTokenizer
         tokenizer = RegexpTokenizer("[\w']+")
         tokenizer.tokenize("Can't is a contraction.")

Out[15]: ["Can't", 'is', 'a', 'contraction']
```

3.4 Filtering stopwords in a tokenized sentence （过滤停词）

```
In [16]: from nltk.corpus import stopwords
         english_stops = set(stopwords.words('english'))
         print(english_stops)

{'yourself', 'what', 'an', 'hasn', 'if', "wouldn't", 'about', "needn't", 'all', 'down', 'and',


In [17]: words = ["Can't", 'is', 'a', 'contraction']

In [18]: [word for word in words if word not in english_stops]

Out[18]: ["Can't", 'contraction']
```

4 Replacing and Correcting Words （替换和纠正）

4.1 Stemming （词干提取）

One of the most common stemming algorithms is the **Porter stemming algorithm** by Martin Porter. It is designed to remove and replace well-known suffixes of English words

```
In [19]: from nltk.stem import PorterStemmer
         stemmer = PorterStemmer()
         stemmer.stem('cooking')

Out[19]: 'cook'
```

4.2 Removing repeating characters （删除重复字符）

In everyday language, people are often not strictly grammatical. They will write things such as I looooooove it in order to emphasize the word love . However, computers don’t know that ”looooooove” is a variation of ”love” unless they are told. This recipe presents a method to remove these annoying repeating characters in order to end up with a proper English word.

```
In [21]: import re
         from nltk.corpus import wordnet

         replacement_patterns = [
             (r'won\'t', 'will not'),
             (r'can\'t', 'cannot'),
             (r'i\'m', 'i am'),
             (r'ain\'t', 'is not'),
             (r'(\w+)\'ll', '\g<1> will'),
             (r'(\w+)n\'t', '\g<1> not'),
             (r'(\w+)\'ve', '\g<1> have'),
             (r'(\w+)\'s', '\g<1> is'),
             (r'(\w+)\'re', '\g<1> are'),
             (r'(\w+)\'d', '\g<1> would')
         ]

         class RegexpReplacer(object):

             def __init__(self, patterns=replacement_patterns):
                 self.patterns = [(re.compile(regex), repl) for (regex, repl) in
                                  patterns]

             def replace(self, text):
                 s = text
                 for (pattern, repl) in self.patterns:
                     (s, count) = re.subn(pattern, repl, s)
                 return s

         class RepeatReplacer(object):

             def __init__(self):
                 self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
                 self.repl = r'\1\2\3'

             def replace(self, word):
                 if wordnet.synsets(word):
                     return word
                 repl_word = self.repeat_regexp.sub(self.repl, word)
                 if repl_word != word:
                     return self.replace(repl_word)
                 else:
                     return repl_word
```

4.3 Spelling correction （拼写纠正）

4.4 Replacing synonyms （同义词替换）

It is often useful to reduce the vocabulary of a text by replacing words with common synonyms. By compressing the vocabulary without losing meaning, you can save memory in cases such as frequency analysis and text indexing. Vocabulary reduction can also increase the occurrence of significant collocations

5 Part-of-speech Tagging （词性标注工具）

5.1 Training a unigram part-of-speech tagger

A unigram generally refers to a single token. Therefore, a unigram tagger only uses a single word as its context for determining the part-of-speech tag.

```
In [23]: from nltk.tag import UnigramTagger
         from nltk.corpus import treebank
         train_sents = treebank.tagged_sents()[:3000]
         tagger = UnigramTagger(train_sents)
         treebank.sents()[0]
```

```
Out[23]: ['Pierre',
          'Vinken',
          ',',
          '61',
          'years',
          'old',
          ',',
          'will',
          'join',
          'the',
          'board',
          'as',
          'a',
          'nonexecutive',
          'director',
          'Nov.',
          '29',
          '.']
```

```
In [24]: tagger.tag(treebank.sents()[0])
```

```
Out[24]: [('Pierre', 'NNP'),
          ('Vinken', 'NNP'),
          (',', ','),
          ('61', 'CD'),
          ('years', 'NNS'),
          ('old', 'JJ'),
          (',', ','),
          ('will', 'MD'),
          ('join', 'VB'),
          ('the', 'DT'),
          ('board', 'NN'),
          ('as', 'IN'),
          ('a', 'DT'),
          ('nonexecutive', 'JJ'),
          ('director', 'NN'),
          ('Nov.', 'NNP'),
          ('29', 'CD'),
          ('.', '.')]
```

5.2 Word Clouds

One approach to visualizing words and counts is word clouds, which artistically lay out the words with sizes proportional to their counts.

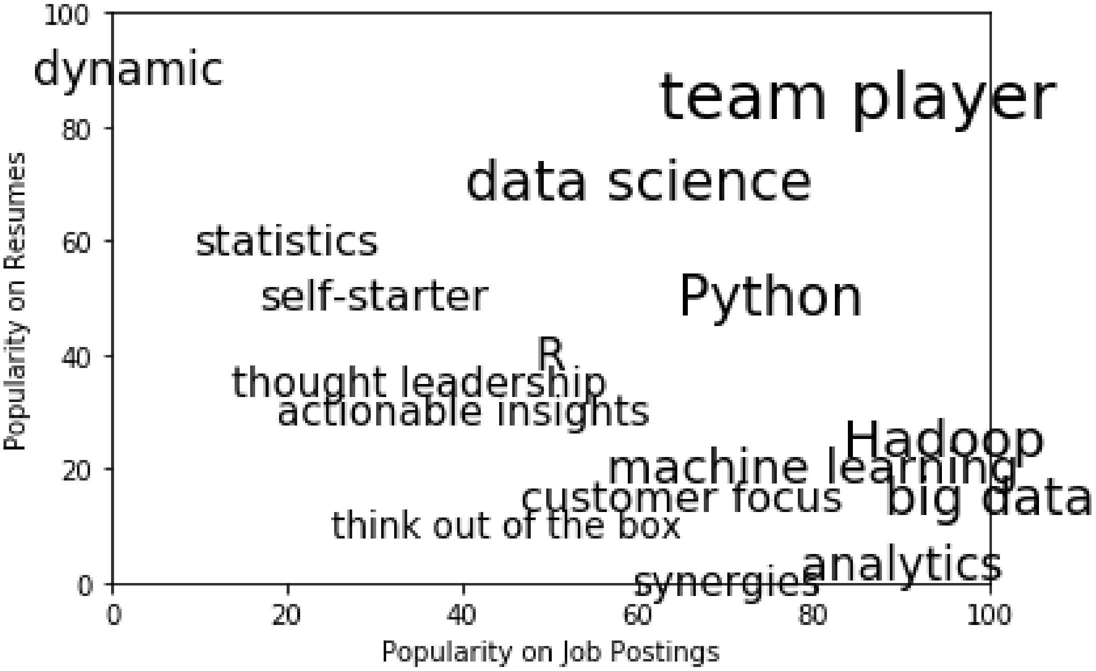
Generally, though, data scientists don't think much of word clouds, in large part because the placement of the words doesn't mean anything other than "here's some space where I was able to fit a word."

If you ever are forced to create a word cloud, think about whether you can make the axes convey something. For example, imagine that, for each of some collection of data science–related buzzwords, you have two numbers between 0 and 100—the first rep- resenting how frequently it appears in job postings, the second how frequently it appears on resumes:

```
In [33]: import matplotlib.pyplot as plt


         def plot_resumes(plt):
             data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
                  ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
                  ("data science", 60, 70), ("analytics", 90, 3),
                  ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
                  ("actionable insights", 40, 30), ("think out of the box", 45, 10),
                  ("self-starter", 30, 50), ("customer focus", 65, 15),
                  ("thought leadership", 35, 35)]

             def text_size(total):
                 """equals 8 if total is 0, 28 if total is 200"""
                 return 8 + total / 200 * 20

             for word, job_popularity, resume_popularity in data:
                 plt.text(job_popularity, resume_popularity, word,
                          ha='center', va='center',
                          size=text_size(job_popularity + resume_popularity))
             plt.xlabel("Popularity on Job Postings")
             plt.ylabel("Popularity on Resumes")
             plt.axis([0, 100, 0, 100])
```

```
    plt.show()
```

```
plot_resumes(plt)
```



6 Text Classification （文本分类）

All are statistics!