**https://www.cnblogs.com/zhbzz2007/p/6177832.html**

[**结巴分词5--关键词抽取**](http://www.cnblogs.com/zhbzz2007/p/6177832.html)

**1 简介**

关键词抽取就是从文本里面把跟这篇文档意义最相关的一些词抽取出来。这个可以追溯到文献检索初期，当时还不支持全文搜索的时候，关键词就可以作为搜索这篇论文的词语。因此，目前依然可以在论文中看到关键词这一项。

除了这些，关键词还可以在文本聚类、分类、自动摘要等领域中有着重要的作用。比如在聚类时将关键词相似的几篇文档看成一个团簇，可以大大提高聚类算法的收敛速度；从某天所有的新闻中提取出这些新闻的关键词，就可以大致了解那天发生了什么事情；或者将某段时间内几个人的微博拼成一篇长文本，然后抽取关键词就可以知道他们主要在讨论什么话题。

总之，关键词就是最能够反映出文本主题或者意思的词语。但是网络上写文章的人不会像写论文那样告诉你本文的关键词是什么，这个时候就需要利用计算机自动抽取出关键词，算法的好坏直接决定了后续步骤的效果。

关键词抽取从方法来说大致有两种：

* 第一种是关键词分配，就是有一个给定的关键词库，然后新来一篇文档，从词库里面找出几个词语作为这篇文档的关键词；
* 第二种是关键词抽取，就是新来一篇文档，从文档中抽取一些词语作为这篇文档的关键词；

目前大多数领域无关的关键词抽取算法（领域无关算法的意思就是无论什么主题或者领域的文本都可以抽取关键词的算法）和它对应的库都是基于后者的。从逻辑上说，后者比前着在实际使用中更有意义。

从算法的角度来看，关键词抽取算法主要有两类：

* 有监督学习算法，将关键词抽取过程视为二分类问题，先抽取出候选词，然后对于每个候选词划定标签，要么是关键词，要么不是关键词，然后训练关键词抽取分类器。当新来一篇文档时，抽取出所有的候选词，然后利用训练好的关键词抽取分类器，对各个候选词进行分类，最终将标签为关键词的候选词作为关键词；
* 无监督学习算法，先抽取出候选词，然后对各个候选词进行打分，然后输出topK个分值最高的候选词作为关键词。根据打分的策略不同，有不同的算法，例如TF-IDF，TextRank等算法；

jieba分词系统中实现了两种关键词抽取算法，分别是基于TF-IDF关键词抽取算法和基于TextRank关键词抽取算法，两类算法均是无监督学习的算法，下面将会通过实例讲解介绍如何使用jieba分词的关键词抽取接口以及通过源码讲解其实现的原理。

**2 示例**

下面将会依次介绍利用jieba分词系统中的TF-IDF及TextRank接口抽取关键词的过程。

**2.1 基于TF-IDF算法进行关键词抽取**

基于TF-IDF算法进行关键词抽取的示例代码如下所示，

from jieba import analyse

# 引入TF-IDF关键词抽取接口

tfidf = analyse.extract\_tags

# 原始文本

text = "线程是程序执行时的最小单位，它是进程的一个执行流，\

是CPU调度和分派的基本单位，一个进程可以由很多个线程组成，\

线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。\

线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。\

同样多线程也可以实现并发操作，每个请求分配一个线程来处理。"

# 基于TF-IDF算法进行关键词抽取

keywords = tfidf(text)

print "keywords by tfidf:"

# 输出抽取出的关键词

for keyword in keywords:

print keyword + "/",

控制台输出，

keywords by tfidf:

线程/ CPU/ 进程/ 调度/ 多线程/ 程序执行/ 每个/ 执行/ 堆栈/ 局部变量/ 单位/ 并发/ 分派/ 一个/ 共享/ 请求/ 最小/ 可以/ 允许/ 分配/

**2.2 基于TextRank算法进行关键词抽取**

基于TextRank算法进行关键词抽取的示例代码如下所示，

from jieba import analyse

# 引入TextRank关键词抽取接口

textrank = analyse.textrank

# 原始文本

text = "线程是程序执行时的最小单位，它是进程的一个执行流，\

是CPU调度和分派的基本单位，一个进程可以由很多个线程组成，\

线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。\

线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。\

同样多线程也可以实现并发操作，每个请求分配一个线程来处理。"

print "\nkeywords by textrank:"

# 基于TextRank算法进行关键词抽取

keywords = textrank(text)

# 输出抽取出的关键词

for keyword in keywords:

print keyword + "/",

控制台输出，

keywords by textrank:

线程/ 进程/ 调度/ 单位/ 操作/ 请求/ 分配/ 允许/ 基本/ 共享/ 并发/ 堆栈/ 独立/ 执行/ 分派/ 组成/ 资源/ 实现/ 运行/ 处理/

**3 理论分析**

下面将会依次分析TF-IDF算法及TextRank算法的原理。

**3.1 TF-IDF算法分析**

在信息检索理论中，TF-IDF是Term Frequency - Inverse Document Frequency的简写。TF-IDF是一种数值统计，用于反映一个词对于语料中某篇文档的重要性。在信息检索和文本挖掘领域，它经常用于因子加权。

TF-IDF的主要思想就是：如果某个词在一篇文档中出现的频率高，也即TF高；并且在语料库中其他文档中很少出现，即DF的低，也即IDF高，则认为这个词具有很好的类别区分能力。

TF-IDF在实际中主要是将二者相乘，也即TF \* IDF，TF为词频（Term Frequency），表示词t在文档d中出现的频率；IDF为反文档频率（Inverse Document Frequency），表示语料库中包含词t的文档的数目的倒数。

**TF公式：**

TF计算公式为，

TF=count(t)count(di)TF=count(t)count(di)

式中，count(t)表示文档di中包含词t的个数；

count(di)表示文档di的词的总数；

**IDF公式：**

IDF计算公式为，

IDF=num(corpus)num(t)+1IDF=num(corpus)num(t)+1

式中，num(corpus)表示语料库corpus中文档的总数；

num(t)表示语料库corpus中包含t的文档的数目；

**应用到关键词抽取：**

1. 预处理，首先进行分词和词性标注，将满足指定词性的词作为候选词；

2. 分别计算每个词的TF-IDF值；

3. 根据每个词的TF-IDF值降序排列，并输出指定个数的词汇作为可能的关键词；

**3.2 TextRank算法分析**

类似于PageRank的思想，将文本中的语法单元视作图中的节点，如果两个语法单元存在一定语法关系（例如共现），则这两个语法单元在图中就会有一条边相互连接，通过一定的迭代次数，最终不同的节点会有不同的权重，权重高的语法单元可以作为关键词。

节点的权重不仅依赖于它的入度结点，还依赖于这些入度结点的权重，入度结点越多，入度结点的权重越大，说明这个结点的权重越高；

TextRank迭代计算公式为，

WS(Vi)=(1−d)+d∗∑Vj∈In(Vi)wji∑Vk∈Out(Vj)wjk∗WS(Vj)WS(Vi)=(1−d)+d∗∑Vj∈In(Vi)wji∑Vk∈Out(Vj)wjk∗WS(Vj)

节点i的权重取决于节点i的邻居节点中i-j这条边的权重 / j的所有出度的边的权重 \* 节点j的权重，将这些邻居节点计算的权重相加，再乘上一定的阻尼系数，就是节点i的权重；

阻尼系数 d 一般取0.85；

**算法通用流程：**

1. 标识文本单元，并将其作为顶点加入到图中；

2. 标识文本单元之间的关系，使用这些关系作为图中顶点之间的边，边可以是有向或者无向，加权或者无权；

3. 基于上述公式，迭代直至收敛；

4. 按照顶点的分数降序排列；

* 1.本模型使用co-occurrence关系，如果两个顶点相应的语义单元共同出现在一个窗口中（窗口大小从2-10不等），那么就连接这两个顶点；
* 2.添加顶点到图中时，需要考虑语法过滤，例如只保留特定词性（如形容词和名词）的词；

**应用到关键短语抽取：**

1. 预处理，首先进行分词和词性标注，将单个word作为结点添加到图中；

2. 设置语法过滤器，将通过语法过滤器的词汇添加到图中；出现在一个窗口中的词汇之间相互形成一条边；

3. 基于上述公式，迭代直至收敛；一般迭代20-30次，迭代阈值设置为0.0001；

4. 根据顶点的分数降序排列，并输出指定个数的词汇作为可能的关键词；

5. 后处理，如果两个词汇在文本中前后连接，那么就将这两个词汇连接在一起，作为关键短语；

**4 源码分析**

jieba分词的关键词抽取功能，是在jieba/analyse目录下实现的。

其中，\_\_init\_\_.py主要用于封装jieba分词的关键词抽取接口；

tfidf.py实现了基于TF-IDF算法抽取关键词；

textrank.py实现了基于TextRank算法抽取关键词；

**4.1 TF-IDF算法抽取关键词源码分析**

基于TF-IDF算法抽取关键词的主调函数是TFIDF.extract\_tags函数，主要是在jieba/analyse/tfidf.py中实现。

其中TFIDF是为TF-IDF算法抽取关键词所定义的类。类在初始化时，默认加载了分词函数tokenizer = jieba.dt、词性标注函数postokenizer = jieba.posseg.dt、停用词stop\_words = self.STOP\_WORDS.copy()、idf词典idf\_loader = IDFLoader(idf\_path or DEFAULT\_IDF)等，并获取idf词典及idf中值（如果某个词没有出现在idf词典中，则将idf中值作为这个词的idf值）。

def \_\_init\_\_(self, idf\_path=None):

# 加载

self.tokenizer = jieba.dt

self.postokenizer = jieba.posseg.dt

self.stop\_words = self.STOP\_WORDS.copy()

self.idf\_loader = IDFLoader(idf\_path or DEFAULT\_IDF)

self.idf\_freq, self.median\_idf = self.idf\_loader.get\_idf()

然后开始通过TF-IDF算法进行关键词抽取。

首先根据是否传入了词性限制集合，来决定是调用词性标注接口还是调用分词接口。例如，词性限制集合为["ns", "n", "vn", "v", "nr"]，表示只能从词性为地名、名词、动名词、动词、人名这些词性的词中抽取关键词。

1） 如果传入了词性限制集合，首先调用词性标注接口，对输入句子进行词性标注，得到分词及对应的词性；依次遍历分词结果，如果该词的词性不在词性限制集合中，则跳过；如果词的长度小于2，或者词为停用词，则跳过；最后将满足条件的词添加到词频词典中，出现的次数加1；然后遍历词频词典，根据idf词典得到每个词的idf值，并除以词频词典中的次数总和，得到每个词的tf \* idf值；如果设置了权重标志位，则根据tf-idf值对词频词典中的词进行降序排序，然后输出topK个词作为关键词；

2） 如果没有传入词性限制集合，首先调用分词接口，对输入句子进行分词，得到分词；依次遍历分词结果，如果词的长度小于2，或者词为停用词，则跳过；最后将满足条件的词添加到词频词典中，出现的次数加1；然后遍历词频词典，根据idf词典得到每个词的idf值，并除以词频词典中的次数总和，得到每个词的tf \* idf值；如果设置了权重标志位，则根据tf-idf值对词频词典中的词进行降序排序，然后输出topK个词作为关键词；

def extract\_tags(self, sentence, topK=20, withWeight=False, allowPOS=(), withFlag=False):

# 传入了词性限制集合

if allowPOS:

allowPOS = frozenset(allowPOS)

# 调用词性标注接口

words = self.postokenizer.cut(sentence)

# 没有传入词性限制集合

else:

# 调用分词接口

words = self.tokenizer.cut(sentence)

freq = {}

for w in words:

if allowPOS:

if w.flag not in allowPOS:

continue

elif not withFlag:

w = w.word

wc = w.word if allowPOS and withFlag else w

# 判断词的长度是否小于2，或者词是否为停用词

if len(wc.strip()) < 2 or wc.lower() in self.stop\_words:

continue

# 将其添加到词频词典中，次数加1

freq[w] = freq.get(w, 0.0) + 1.0

# 统计词频词典中的总次数

total = sum(freq.values())

for k in freq:

kw = k.word if allowPOS and withFlag else k

# 计算每个词的tf-idf值

freq[k] \*= self.idf\_freq.get(kw, self.median\_idf) / total

# 根据tf-idf值进行排序

if withWeight:

tags = sorted(freq.items(), key=itemgetter(1), reverse=True)

else:

tags = sorted(freq, key=freq.\_\_getitem\_\_, reverse=True)

# 输出topK个词作为关键词

if topK:

return tags[:topK]

else:

return tags

**4.2 TextRank算法抽取关键词源码分析**

基于TextRank算法抽取关键词的主调函数是TextRank.textrank函数，主要是在jieba/analyse/textrank.py中实现。

其中，TextRank是为TextRank算法抽取关键词所定义的类。类在初始化时，默认加载了分词函数和词性标注函数tokenizer = postokenizer = jieba.posseg.dt、停用词表stop\_words = self.STOP\_WORDS.copy()、词性过滤集合pos\_filt = frozenset(('ns', 'n', 'vn', 'v'))，窗口span = 5，(("ns", "n", "vn", "v"))表示词性为地名、名词、动名词、动词。

首先定义一个无向有权图，然后对句子进行分词；依次遍历分词结果，如果某个词i满足过滤条件（词性在词性过滤集合中，并且词的长度大于等于2，并且词不是停用词），然后将这个词之后窗口范围内的词j（这些词也需要满足过滤条件），将它们两两（词i和词j）作为key，出现的次数作为value，添加到共现词典中；

然后，依次遍历共现词典，将词典中的每个元素，key = （词i，词j），value = 词i和词j出现的次数，其中词i，词j作为一条边起始点和终止点，共现的次数作为边的权重，添加到之前定义的无向有权图中。

然后对这个无向有权图进行迭代运算textrank算法，最终经过若干次迭代后，算法收敛，每个词都对应一个指标值；

如果设置了权重标志位，则根据指标值值对无向有权图中的词进行降序排序，最后输出topK个词作为关键词；

def textrank(self, sentence, topK=20, withWeight=False, allowPOS=('ns', 'n', 'vn', 'v'), withFlag=False):

self.pos\_filt = frozenset(allowPOS)

# 定义无向有权图

g = UndirectWeightedGraph()

# 定义共现词典

cm = defaultdict(int)

# 分词

words = tuple(self.tokenizer.cut(sentence))

# 依次遍历每个词

for i, wp in enumerate(words):

# 词i 满足过滤条件

if self.pairfilter(wp):

# 依次遍历词i 之后窗口范围内的词

for j in xrange(i + 1, i + self.span):

# 词j 不能超出整个句子

if j >= len(words):

break

# 词j不满足过滤条件，则跳过

if not self.pairfilter(words[j]):

continue

# 将词i和词j作为key，出现的次数作为value，添加到共现词典中

if allowPOS and withFlag:

cm[(wp, words[j])] += 1

else:

cm[(wp.word, words[j].word)] += 1

# 依次遍历共现词典的每个元素，将词i，词j作为一条边起始点和终止点，共现的次数作为边的权重

for terms, w in cm.items():

g.addEdge(terms[0], terms[1], w)

# 运行textrank算法

nodes\_rank = g.rank()

# 根据指标值进行排序

if withWeight:

tags = sorted(nodes\_rank.items(), key=itemgetter(1), reverse=True)

else:

tags = sorted(nodes\_rank, key=nodes\_rank.\_\_getitem\_\_, reverse=True)

# 输出topK个词作为关键词

if topK:

return tags[:topK]

else:

return tags

其中，无向有权图的的定义及实现是在UndirectWeightedGraph类中实现的。根据UndirectWeightedGraph类的初始化函数\_\_init\_\_，我们可以发现，所谓的无向有权图就是一个词典，词典的key是后续要添加的词，词典的value，则是一个由（起始点，终止点，边的权重）构成的三元组所组成的列表，表示以这个词作为起始点的所有的边。

无向有权图添加边的操作是在addEdge函数中完成的，因为是无向图，所以我们需要依次将start作为起始点，end作为终止点，然后再将start作为终止点，end作为起始点，这两条边的权重是相同的。

def addEdge(self, start, end, weight):

# use a tuple (start, end, weight) instead of a Edge object

self.graph[start].append((start, end, weight))

self.graph[end].append((end, start, weight))

执行textrank算法迭代是在rank函数中完成的。

首先对每个结点赋予相同的权重，以及计算出该结点的所有出度的次数之和；

然后迭代若干次，以确保得到稳定的结果；

在每一次迭代中，依次遍历每个结点；对于结点n，首先根据无向有权图得到结点n的所有  
入度结点（对于无向有权图，入度结点与出度结点是相同的，都是与结点n相连的结点），在前面我们已经计算出这个入度结点的所有出度的次数，而它对于结点n的权值的贡献等于它本身的权值 乘以 它与结点n的共现次数 / 这个结点的所有出度的次数 ，将各个入度结点得到的权值相加，再乘以一定的阻尼系数，即可得到结点n的权值；

迭代完成后，对权值进行归一化，并返回各个结点及其对应的权值。

def rank(self):

ws = defaultdict(float)

outSum = defaultdict(float)

wsdef = 1.0 / (len(self.graph) or 1.0)

# 初始化各个结点的权值

# 统计各个结点的出度的次数之和

for n, out in self.graph.items():

ws[n] = wsdef

outSum[n] = sum((e[2] for e in out), 0.0)

# this line for build stable iteration

sorted\_keys = sorted(self.graph.keys())

# 遍历若干次

for x in xrange(10): # 10 iters

# 遍历各个结点

for n in sorted\_keys:

s = 0

# 遍历结点的入度结点

for e in self.graph[n]:

# 将这些入度结点贡献后的权值相加

# 贡献率 = 入度结点与结点n的共现次数 / 入度结点的所有出度的次数

s += e[2] / outSum[e[1]] \* ws[e[1]]

# 更新结点n的权值

ws[n] = (1 - self.d) + self.d \* s

(min\_rank, max\_rank) = (sys.float\_info[0], sys.float\_info[3])

# 获取权值的最大值和最小值

for w in itervalues(ws):

if w < min\_rank:

min\_rank = w

if w > max\_rank:

max\_rank = w

# 对权值进行归一化

for n, w in ws.items():

# to unify the weights, don't \*100.

ws[n] = (w - min\_rank / 10.0) / (max\_rank - min\_rank / 10.0)

return ws

**4.3 使用自定义停用词集合**

jieba分词中基于TF-IDF算法抽取关键词以及基于TextRank算法抽取关键词均需要利用停用词对候选词进行过滤。实现TF-IDF算法抽取关键词的类TFIDF和实现TextRank算法抽取关键词的类TextRank都是类KeywordExtractor的子类。而在类KeywordExtractor，实现了一个方法，可以根据用户指定的路径，加载用户提供的停用词集合。

类KeywordExtractor是在jieba/analyse/tfidf.py中实现。

类KeywordExtractor首先提供了一个默认的名为STOP\_WORDS的停用词集合。

然后，类KeywordExtractor实现了一个方法set\_stop\_words，可以根据用户指定的路径，加载用户提供的停用词集合。

可以将extra\_dict/stop\_words.txt拷贝出来，并在文件末尾两行分别加入“一个”和  
“每个”这两个词，作为用户提供的停用词文件，使用用户提供的停用词集合进行关键词抽取的实例代码如下，

from jieba import analyse

# 引入TF-IDF关键词抽取接口

tfidf = analyse.extract\_tags

# 使用自定义停用词集合

analyse.set\_stop\_words("stop\_words.txt")

# 原始文本

text = "线程是程序执行时的最小单位，它是进程的一个执行流，\

是CPU调度和分派的基本单位，一个进程可以由很多个线程组成，\

线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。\

线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。\

同样多线程也可以实现并发操作，每个请求分配一个线程来处理。"

# 基于TF-IDF算法进行关键词抽取

keywords = tfidf(text)

print "keywords by tfidf:"

# 输出抽取出的关键词

for keyword in keywords:

print keyword + "/",

关键词结果为，

keywords by tfidf:

线程/ CPU/ 进程/ 调度/ 多线程/ 程序执行/ 执行/ 堆栈/ 局部变量/ 单位/ 并发/ 分派/ 共享/ 请求/ 最小/ 可以/ 允许/ 分配/ 多个/ 运行/

对比章节2.1中的关键词抽取结果，可以发现“一个”和“每个”这两个词没有抽取出来。

keywords by tfidf:

线程/ CPU/ 进程/ 调度/ 多线程/ 程序执行/ 每个/ 执行/ 堆栈/ 局部变量/ 单位/ 并发/ 分派/ 一个/ 共享/ 请求/ 最小/ 可以/ 允许/ 分配/

**实现原理** ，这里仍然以基于TF-IDF算法抽取关键词为例。

前面已经介绍了，jieba/analyse/\_\_init\_\_.py主要用于封装jieba分词的关键词抽取接口，在\_\_init\_\_.py首先将类TFIDF实例化为对象default\_tfidf，而类TFIDF在初始化时会设置停用词表，我们知道类TFIDF是类KeywordExtractor的子类，而类KeywordExtractor中提供了一个名为STOP\_WORDS的停用词集合，因此类TFIDF在初始化时先将类KeywordExtractor中的STOP\_WORDS拷贝过来，作为自己的停用词集合stop\_words。

# 实例化TFIDF类

default\_tfidf = TFIDF()

# 实例化TextRank类

default\_textrank = TextRank()

extract\_tags = tfidf = default\_tfidf.extract\_tags

set\_idf\_path = default\_tfidf.set\_idf\_path

textrank = default\_textrank.extract\_tags

# 用户设置停用词集合接口

def set\_stop\_words(stop\_words\_path):

# 更新对象default\_tfidf中的停用词集合

default\_tfidf.set\_stop\_words(stop\_words\_path)

# 更新对象default\_textrank中的停用词集合

default\_textrank.set\_stop\_words(stop\_words\_path)

如果用户需要使用自己提供的停用词集合，则需要调用analyse.set\_stop\_words(stop\_words\_path)这个函数，set\_stop\_words函数是在类KeywordExtractor实现的。set\_stop\_words函数执行时，会更新对象default\_tfidf中的停用词集合stop\_words，当set\_stop\_words函数执行完毕时，stop\_words也就是更新后的停用词集合。我们可以做个实验，验证在调用analyse.set\_stop\_words(stop\_words\_path)函数前后，停用词集合是否发生改变。

from jieba import analyse

import copy

# 将STOP\_WORDS集合深度拷贝出来

stopwords0 = copy.deepcopy(analyse.default\_tfidf.STOP\_WORDS)

# 设置用户自定停用词集合之前，将停用词集合深度拷贝出来

stopwords1 = copy.deepcopy(analyse.default\_tfidf.stop\_words)

print stopwords0 == stopwords1

print stopwords1 - stopwords0

# 设置用户自定停用词集合

analyse.set\_stop\_words("stop\_words.txt")

# 设置用户自定停用词集合之后，将停用词集合深度拷贝出来

stopwords2 = copy.deepcopy(analyse.default\_tfidf.stop\_words)

print stopwords1 == stopwords2

print stopwords2 - stopwords1

结果如下所示，

True

set([])

False

set([u'\u6bcf\u4e2a', u'\u8207', u'\u4e86', u'\u4e00\u500b', u'\u800c', u'\u4ed6\u5011', u'\u6216', u'\u7684', u'\u4e00\u4e2a', u'\u662f', u'\u5c31', u'\u4f60\u5011', u'\u5979\u5011', u'\u6c92\u6709', u'\u57fa\u672c', u'\u59b3\u5011', u'\u53ca', u'\u548c', u'\u8457', u'\u6211\u5011', u'\u662f\u5426', u'\u90fd'])

说明：

* 没有加载用户提供的停用词集合之前，停用词集合就是类KeywordExtractor中的STOP\_WORDS拷贝过来的；
* 加载用户提供的停用词集合之后，停用词集合在原有的基础上进行了扩；

证明了我们的想法。