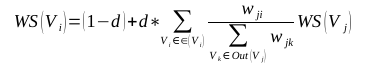
# TextRank

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。参考论文：Rada Mihalcea的《TextRank：Bring Order into texts》。

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；图中任两点 Vi , Vj 之间边的权重为 wji , 对于一个给定的点 Vi, In(Vi) 为 指 向 该 点 的 点 集 合 , Out(Vi) 为点 Vi 指向的点集合。

TextRank迭代计算公式为，



其中, d 为阻尼系数, 取值范围为 0 到 1, 代表从图中某一特定点指向其他任意点的概率, 一般取值为 0.85。使用TextRank 算法计算图中各点的得分时, 需要给图中的点指定任意的初值, 并递归计算直到收敛, 即图中任意一点的误差率小于给定的极限值时就可以达到收敛, 一般该极限值取 0.0001。算法通用流程：

**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；

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长度为5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