# 1英文分词

<http://itindex.net/detail/48243-%E8%8B%B1%E6%96%87-%E5%88%86%E8%AF%8D-%E7%AE%97%E6%B3%95?utm_source=tuicool&utm_medium=referral>

根据文档相关性计算公式

* TF-IDF: <http://lutaf.com/210.htm>
* BM25: <http://lutaf.com/211.htm>

分词质量对于基于词频的相关性计算是无比重要的

英文(西方语言）语言的基本单位就是单词，所以分词特别容易做，只需要3步：

1. 根据空格/符号/段落 分隔,得到单词组
2. 过滤，排除掉stop word
3. 提取词干

### 第一步：按空格/符号分词

用正则表达式很容易

pattern = r'''(?x) # set flag to allow verbose regexps

([A-Z]\.)+ # abbreviations, e.g. U.S.A.

| \w+(-\w+)\* # words with optional internal hyphens

| \$?\d+(\.\d+)?%? # currency and percentages, e.g. $12.40, 82%

| \.\.\. # ellipsis

| [][.,;"'?():-\_`] # these are separate tokens

'''

re.findall(pattern,待分词文本)

### 第二步：排除stop word

stopword就是类似 a/an/and/are/then 的这类高频词，高频词会对基于词频的算分公式产生极大的干扰，所以需要过滤

### 第三步：提取词干

词干提取( **Stemming**) 这是西方语言特有的处理，比如说英文单词有 单数复数的变形，-ing和-ed的变形，但是在计算相关性的时候，应该当做同一个单词。比如 apple和apples，doing和done是同一个词，提取词干的目的就是要合并这些变态

Stemming有3大主流算法

* [Porter Stemming](http://lutaf.com/j?u=http://www.tartarus.org/~martin/PorterStemmer/index.html)
* [Lovins stemmer](http://lutaf.com/j?u=http://www.cs.waikato.ac.nz/~eibe/stemmers/index.html)
* [Lancaster Stemming](http://lutaf.com/j?u=http://www.comp.lancs.ac.uk/computing/research/stemming/index.htm)

Lucene 英文分词自带了3个stemming算法，分别是

1. EnglishMinimalStemmer
2. 著名的 Porter Stemming
3. KStemmer

词干提取算法并不复杂，要么是一堆规则，要么用映射表，编程容易，但是必须是这种语言的专家，了解构词法才行啊

<http://text-processing.com/demo/stem/> 是一个在线试验词干提取算法的网站

### Lemmatisation

Lemmatisation是和词干提取(Stemming) 齐名的一个语言学名词，中文可以叫做 **词形还原** ,就是通过查询字典，把 "drove" 还原到 "drive"   
而stemming会把单词变短，"apples","apple"处理之后都变成了 "appl"

* [wikipedia关于词形还原的简介](http://lutaf.com/j?u=http://en.wikipedia.org/wiki/Lemmatisation)
* [European languages lemmatizer](http://lutaf.com/j?u=http://lemmatizer.org/) 一个c语言的lib

做计算机语言学研究才会涉及到lemmatization，我个人觉得做搜索完全可以不考虑，Stemming已经可以解决大问题了

**参考**

* [http://text-processing.com/](http://lutaf.com/j?u=http://text-processing.com/)
* [www.nltk.org python的自然语言包，非常有用](http://lutaf.com/j?u=http://www.nltk.org)
* [PYTHON自然语言处理中文版.pdf](http://lutaf.com/j?u=http://vdisk.weibo.com/s/4ffue/1334656530)

# 2中文分词

暂无

# 3用python做文本挖掘的流程

* 收集数据
  + 数据集。如果是已经被人做成数据集了，这就省去了很多麻烦事
  + 抓取。这个是 Python 做得最好的事情，优秀的包有很多，比如 [scrapy](https://link.zhihu.com/?target=http://scrapy.org" \t "https://zhuanlan.zhihu.com/p/_blank)，[beautifulsoup](https://link.zhihu.com/?target=http://www.crummy.com/software/BeautifulSoup/" \t "https://zhuanlan.zhihu.com/p/_blank) 等等。
* 预处理（对[这里](https://link.zhihu.com/?target=https://lists.cs.princeton.edu/pipermail/topic-models/2013-November/002593.html" \t "https://zhuanlan.zhihu.com/p/_blank)的高质量讨论结果的修改，下面的顺序仅限**英文**）

1. 去掉抓来的数据中不需要的部分，比如 HTML TAG，只保留文本。结合 beautifulsoup 和正则表达式就可以了。[pattern.web](https://link.zhihu.com/?target=http://www.clips.ua.ac.be/pages/pattern-web" \t "https://zhuanlan.zhihu.com/p/_blank) 也有相关功能。
2. 处理编码问题。没错，即使是英文也需要处理编码问题！由于 Python2 的历史原因，不得不在编程的时候自己处理。英文也存在 unicode 和 utf-8 转换的问题，中文以及其他语言就更不用提了。[这里](https://link.zhihu.com/?target=http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6095/pdf" \t "https://zhuanlan.zhihu.com/p/_blank)有一个讨论，可以参考，当然网上也有很多方案，找到一个适用于自己的最好。
3. 将文档分割成句子。
4. 将句子分割成词。专业的叫法是 tokenize。
5. 拼写错误纠正。[pyenchant](https://link.zhihu.com/?target=http://pythonhosted.org/pyenchant/" \t "https://zhuanlan.zhihu.com/p/_blank) 可以帮你！（中文就没有这么些破事！）
6. POS Tagging。nltk 是不二选择，还可以使用 [pattern](https://link.zhihu.com/?target=http://www.clips.ua.ac.be/pages/pattern" \t "https://zhuanlan.zhihu.com/p/_blank)。
7. 去掉标点符号。使用正则表达式就可以。
8. 去掉长度过小的单词。len<3 的是通常选择。
9. 去掉 non-alpha 词。同样，可以用正则表达式完成 \W 就可以。
10. 转换成小写。
11. 去掉停用词。Matthew L. Jockers 提供了一份比机器学习和自然语言处理中常用的停词表[更长的停词表](https://link.zhihu.com/?target=http://www.matthewjockers.net/wp-content/uploads/2013/04/uwm-workshop.zip" \t "https://zhuanlan.zhihu.com/p/_blank)。[中文的停词表](https://link.zhihu.com/?target=http://www.datatang.com/data/43894" \t "https://zhuanlan.zhihu.com/p/_blank) 可以参考这个。
12. lemmatization/stemming。nltk 里面提供了好多种方式，推荐用 wordnet 的方式，这样不会出现把词过分精简，导致词丢掉原型的结果，如果实在不行，也用 snowball 吧，别用 porter，porter 的结果我个人太难接受了，弄出结果之后都根本不知道是啥词了。[MBSP](https://link.zhihu.com/?target=http://www.clips.ua.ac.be/pages/MBSP" \t "https://zhuanlan.zhihu.com/p/_blank) 也有相关功能。
13. 重新去掉长度过小的词。是的，再来一遍。
14. 重新去停词。上面这两部完全是为了更干净。
15. 到这里拿到的基本上是非常干净的文本了。如果还有进一步需求，还可以根据 POS 的结果继续选择某一种或者几种词性的词。

* [Bag-of-Words](https://link.zhihu.com/?target=http://weibo.com/1657470871/Aicj2oP8G" \t "https://zhuanlan.zhihu.com/p/_blank)! nltk 和 [scikit.learn](https://link.zhihu.com/?target=http://scikit-learn.org/stable/" \t "https://zhuanlan.zhihu.com/p/_blank) 里面都有很完整的方案，自己选择合适的就好。这里[如果不喜欢没有次序的 unigram 模型](https://link.zhihu.com/?target=http://scikit-learn.org/stable/modules/feature_extraction.html%23limitations-of-the-bag-of-words-representation" \t "https://zhuanlan.zhihu.com/p/_blank)，可以自行选择 bi-gram 和 tri-gram 以及更高的 n-gram 模型。nltk 和 sklearn里面都有相关的处理方法。
* 更高级的特征。
  + TF-IDF。这个 nltk 和 sklearn 里面也都有。
  + [Hashing](https://link.zhihu.com/?target=http://scikit-learn.org/stable/modules/feature_extraction.html%23vectorizing-a-large-text-corpus-with-the-hashing-trick" \t "https://zhuanlan.zhihu.com/p/_blank)！
* 训练模型
  + 到这里，就根据自己的应用选择合适的学习器就好了。
  + 分类，情感分析等。[sklearn](https://link.zhihu.com/?target=http://scikit-learn.org/stable/supervised_learning.html" \t "https://zhuanlan.zhihu.com/p/_blank) 里面很多方法，pattern 里有[情感分析的模块](https://link.zhihu.com/?target=http://www.clips.ua.ac.be/pages/pattern-en%23sentiment" \t "https://zhuanlan.zhihu.com/p/_blank)，nltk 中也有一些分类器。
  + 主题发现
    - [NMF](https://link.zhihu.com/?target=http://scikit-learn.org/stable/auto_examples/applications/topics_extraction_with_nmf.html%23example-applications-topics-extraction-with-nmf-py" \t "https://zhuanlan.zhihu.com/p/_blank)
    - [(Online) Latent Dirichlet Allocation](https://link.zhihu.com/?target=http://radimrehurek.com/gensim/wiki.html%23latent-dirichlet-allocation" \t "https://zhuanlan.zhihu.com/p/_blank)
    - [word2vec](https://link.zhihu.com/?target=http://radimrehurek.com/gensim/models/word2vec.html" \t "https://zhuanlan.zhihu.com/p/_blank)
  + 自动文摘。这个自己写吧，没发现什么成型的工具。
* Draw results
  + Matplotlib
  + Tag cloud
  + [Graph](https://link.zhihu.com/?target=http://www.clips.ua.ac.be/pages/pattern-graph" \t "https://zhuanlan.zhihu.com/p/_blank)

**----------更新分割线 2013.12.06 --------------**

说明：在预处理部分仅仅针对英文的情况，由于中英文在分词上是不同的机制，所以在处理中文的时候需要根据情况进行，个人经验是在去停词之前分词。这部分有待完善。

* 中文分词。[jieba](https://link.zhihu.com/?target=https://github.com/fxsjy/jieba" \t "https://zhuanlan.zhihu.com/p/_blank) 或者 [Stanford (Chinese) Word Segmenter](https://link.zhihu.com/?target=http://nlp.stanford.edu/software/segmenter.shtml" \t "https://zhuanlan.zhihu.com/p/_blank)。jieba 是纯 Python 写的，Stanford 的可以通过 [nltk](https://link.zhihu.com/?target=http://nltk.org/" \t "https://zhuanlan.zhihu.com/p/_blank) 调用，[复旦 NLP](https://link.zhihu.com/?target=http://code.google.com/p/fudannlp/" \t "https://zhuanlan.zhihu.com/p/_blank) 也可以用 Python 调用。

# 4怎样写一个拼写检查器（实际主要是纠正功能，即找到实际正确的单词）

<http://blog.sina.com.cn/s/blog_617c80460100z3sc.html>

上个星期, 我的两个朋友 Dean 和 Bill 分别告诉我说他们对 Google 的快速高质量的拼写检查工具感到惊奇. 比如说在搜索的时候键入 [speling], 在不到 0.1 秒的时间内, Google 会返回: 你要找的是不是 [spelling]. (Yahoo! 和 微软也有类似的功能). 让我感到有点奇怪的是我原想 Dean 和 Bill 这两个很牛的工程师和数学家应该对于使用统计语言模型构建拼写检查器有职业的敏感. 但是他们似乎没有这个想法. 我后来想了想, 他们的确没什么理由很熟悉统计语言模型. 不是他们的知识有问题, 而是我预想的本来就是不对的.  
  
我觉得, 如果对这方面的工作做个解释, 他们和其他人肯定会受益. 然而像Google 的那样工业强度的拼写检查器的全部细节只会让人感到迷惑而不是受到启迪. 前几天我乘飞机回家的时候, 顺便写了几十行程序, 作为一个玩具性质的拼写检查器. 这个拼写检查器大约1秒能处理10多个单词, 并且达到 80% -90% 的准确率. 下面就是我的代码, 用Python 2.5 写成, 一共21 行, 是一个功能完备的拼写检查器.

import re, collections  
  
def words(text): return re.findall('[a-z]+', text.lower())   
  
def train(features):

    model = collections.defaultdict(lambda: 1)

    for f in features:

        model[f] += 1

    return model

NWORDS = train(words(file('big.txt').read()))

alphabet = 'abcdefghijklmnopqrstuvwxyz'  
  
def edits1(word):

    n = len(word)

    return set([word[0:i]+word[i+1:] for i in range(n)] +                     # deletion

               [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] + # transposition

               [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] + # alteration

               [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])  # insertion  
  
def known\_edits2(word):

    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)  
  
def known(words): return set(w for w in words if w in NWORDS)  
  
def correct(word):

    candidates = known([word]) or known(edits1(word)) or known\_edits2(word) or [word]

    return max(candidates, key=lambda w: NWORDS[w])

这段代码定义了一个函数叫 correct, 它以一个单词作为输入参数, 返回最可能的拼写建议结果. 比如说:

>>> correct('speling')  
'spelling'  
>>> correct('korrecter')  
'corrector'

## 拼写检查器的原理, 一些简单的概率知识

我简单的介绍一下它的工作原理. 给定一个单词, 我们的任务是选择和它最相似的拼写正确的单词. (如果这个单词本身拼写就是正确的, 那么最相近的就是它自己啦). 当然, 不可能绝对的找到相近的单词, 比如说给定 lates 这个单词, 它应该别更正为 late 呢 还是 latest 呢? 这些困难指示我们, 需要使用概率论, 而不是基于规则的判断. 我们说, 给定一个词 w, 在所有正确的拼写词中, 我们想要找一个正确的词 c, 使得对于 w 的条件概率最大, 也就是说:

argmax*c* P(*c*|*w*)

按照 [贝叶斯理论](http://en.wikipedia.org/wiki/Bayes'_theorem) 上面的式子等价于:

argmax*c* P(*w*|*c*) P(*c*) / P(*w*)

因为用户可以输错任何词, 因此对于任何 c 来讲, 出现 w 的概率 P(w) 都是一样的, 从而我们在上式中忽略它, 写成:

argmax*c* P(*w*|*c*) P(*c*)

这个式子有三个部分, 从右到左, 分别是:

1. P(c), 文章中出现一个正确拼写词 c 的概率, 也就是说, 在英语文章中, c 出现的概率有多大呢? 因为这个概率完全由英语这种语言决定, 我们称之为做语言模型. 好比说, 英语中出现 the 的概率  P('the') 就相对高, 而出现  P('zxzxzxzyy') 的概率接近0(假设后者也是一个词的话).  
  
2. P(w|c), 在用户想键入 c 的情况下敲成 w 的概率. 因为这个是代表用户会以多大的概率把 c 敲错成 w, 因此这个被称为误 差模型.  
  
3. argmax*c*, 用来枚举所有可能的 c 并且选取概率最大的, 因为我们有理由相信, 一个(正确的)单词出现的频率高, 用户又容易把它敲成另一个错误的单词, 那么, 那个敲错的单词应该被更正为这个正确的.

有人肯定要问, 你笨啊, 为什么把最简单的一个 P(*c*|*w*) 变成两项复杂的式子来计算? 答案是本质上 P(c|w) 就是和这两项同时相关的, 因此拆成两项反而容易处理. 举个例子, 比如一个单词 thew 拼错了. 看上去 thaw 应该是正确的, 因为就是把 a 打成 e 了. 然而, 也有可能用户想要的是 the, 因为 the 是英语中常见的一个词, 并且很有可能打字时候手不小心从 e 滑到 w 了. 因此, 在这种情况下, 我们想要计算  P(*c*|*w*), 就必须同时考虑 c 出现的概率和从 c 到 w 的概率. 把一项拆成两项反而让这个问题更加容易更加清晰.  
  
现在, 让我们看看程序究竟是怎么一回事. 首先是计算 P(c), 我们可以读入一个巨大的文本文件, [big.txt](http://www.norvig.com/big.txt), 这个里面大约有几百万个词(相当于是语料库了). 这个文件是由[Gutenberg 计划](http://www.gutenberg.org/wiki/Main_Page) 中可以获取的一些书, [Wiktionary](http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists) 和 [British National Corpus](http://www.kilgarriff.co.uk/bnc-readme.html) 语料库构成. (当时在飞机上我只有福尔摩斯全集, 我后来又加入了一些, 直到效果不再显著提高为止).  
  
然后, 我们利用一个叫 words 的函数把语料中的单词全部抽取出来, 转成小写, 并且去除单词中间的特殊符号. 这样, 单词就会成为字母序列, don't 就变成 don 和 t 了.1 接着我们训练一个概率模型, 别被这个术语吓倒, 实际上就是数一数每个单词出现几次. 在 train 函数中, 我们就做这个事情.

def words(text): return re.findall('[a-z]+', text.lower())   
  
def train(features):

    model = collections.defaultdict(lambda: 1)

    for f in features:

        model[f] += 1

    return model

NWORDS = train(words(file('big.txt').read()))

实际上, NWORDS[w] 存储了单词 w 在语料中出现了多少次. 不过一个问题是要是遇到我们从来没有过见过的新词怎么办. 假如说一个词拼写完全正确, 但是语料库中没有包含这个词, 从而这个词也永远不会出现在训练集中. 于是, 我们就要返回出现这个词的概率是0. 这个情况不太妙, 因为概率为0这个代表了这个事件绝对不可能发生, 而在我们的概率模型中, 我们期望用一个很小的概率来代表这种情况. 实际上处理这个问题有很多成型的标准方法, 我们选取一个最简单的方法: 从来没有过见过的新词一律假设出现过一次. 这个过程一般成为”平滑化”, 因为我们把概率分布为0的设置为一个小的概率值. 在语言实现上, 我们可以使用Python collention 包中的 defaultdict 类, 这个类和 python 标准的 dict (其他语言中可能称之为 hash 表) 一样, 唯一的不同就是可以给任意的键设置一个默认值, 在我们的例子中, 我们使用一个匿名的 lambda:1 函数, 设置默认值为 1.

然后的问题是: 给定一个单词 w, 怎么能够枚举所有可能的正确的拼写呢? 实际上前人已经研究得很充分了, 这个就是一个**编辑距离**的概 念. 这两个词之间的编辑距离  
定义为使用了几次插入(在词中插入一个单字母), 删除(删除一个单字母), 交换(交换相邻两个字母), 替换(把一个字母换成另一个)的操作从一个词变到另一个词.  
下面这个函数可以返回所有与单词 w 编辑距离为 1 的集合.

def edits1(word):

    n = len(word)

    return set([word[0:i]+word[i+1:] for i in range(n)] +                     # deletion

               [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] + # transposition

               [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] + # alteration

               [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])  # insertion

显然, 这个集合很大. 对于一个长度为 n 的单词, 可能有n种删除, n-1中对换, 26n 种 (译注: 实际上是 25n 种)替换 和 26(n+1) 种插入 (译注: 实际上比这个小, 因为在一个字母前后再插入这个字母构成的词是等价的). 这样的话, 一共就是 54n + 25 中情况 (当中还有一点重复). 比如说, 和 something 这个单词的编辑距离为1 的词按照这个算来是 511 个, 而实际上是 494 个.

一般讲拼写检查的文献宣称大约80-95%的拼写错误都是介于编译距离 1 以内. 然而下面我们看到, 当我对于一个有270个拼写错误的语料做实验的时候, 我发现只有76%的拼写错误是属于编辑距离为1的集合. 或许是我选取的例子比典型的例子难处理一点吧. 不管怎样, 我觉得这个结果不够好, 因此我开始考虑编辑距离为 2 的那些单词了. 这个事情很简单, 递归的来看, 就是把 edit1 函数再作用在 edit1 函数的返回集合的每一个元素上就行了. 因此, 我们定义函数 edit2:

def edits2(word):

    return set(e2 for e1 in edits1(word) for e2 in edits1(e1))

这个语句写起来很简单, 实际上背后是很庞大的计算量: 与 something 编辑距离为2的单词居然达到了 114,324 个. 不过编辑距离放宽到2以后, 我们基本上就能覆盖所有的情况了, 在270个样例中, 只有3个的编辑距离大于2. 当然我们可以做一些小小的优化: 在这些编辑距离小于2的词中间, 只把那些正确的词作为候选词. 我们仍然考虑所有的可能性, 但是不需要构建一个很大的集合, 因此, 我们构建一个函数叫做 known\_edits2, 这个函数只返回那些正确的并且与 w 编辑距离小于2 的词的集合:

def known\_edits2(word):

    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

现在, 在刚才的 something 例子中, known\_edits2('something') 只能返回 3 个单词: 'smoothing', 'something' 和 'soothing', 而实际上所有编辑距离为 1 或者  2 的词一共有 114,324 个. 这个优化大约把速度提高了 10%.

最后剩下的就是误差模型部分 P(*w*|*c*) 了. 这个也是当时难住我的部分. 当时我在飞机上, 没有网络, 也就没有数据用来构建一个拼写错误模型. 不过我有一些常识性的知识: 把一个元音拼成另一个的概率要大于辅音 (因为人常常把 hello 打成 hallo 这样); 把单词的第一个字母拼错的概率会相对小, 等等. 但是我并没有具体的数字去支撑这些证据. 因此, 我选择了一个简单的方法: 编辑距离为1的正确单词比编辑距离为2的优先级高, 而编辑距离为0的正确单词优先级比编辑距离为1的高. 因此, 用代码写出来就是:

(译注: 此处作者使用了Python语言的一个巧妙性质: 短路表达式. 在下面的代码中, 如果known(set)非空, candidate 就会选取这个集合, 而不继续计算后面的; 因此, 通过Python语言的短路表达式, 作者很简单的实现了优先级)

def known(words): return set(w for w in words if w in NWORDS)  
  
def correct(word):

    candidates = known([word]) or known(edits1(word)) or known\_edits2(word) or [word]

    return max(candidates, key=lambda w: NWORDS[w])

correct 函数从一个候选集合中选取最大概率的. 实际上, 就是选取有最大 P(*c*) 值的那个. 所有的 P(c) 值都存储在 NWORDS 结构中.

## 效果

现在我们看看算法效果怎么样. 在飞机上我尝试了好几个例子, 效果还行. 飞机着陆后, 我从牛津文本档案库 (Oxford Text Archive)下载了 Roger Mitton 的 [Birkbeck 拼写错误语料库](http://ota.ahds.ac.uk/texts/0643.html). 从这个库中, 我取出了两个集合, 作为我要做拼写检查的目标. 第一个集合用来作为在开发中作为参考, 第二个作为最后的结果测试. 也就是说, 我程序完成之前不参考它, 而把程序在其上的测试结果作为最后的效果. 用两个集合一个训练一个对照是一种良好的实践, 至少这样可以避免我通过对特定数据集合进行特殊调整从而自欺欺人. 这里我给出了一个测试的例子和一个运行测试的例子. 实际的完整测试例子和程序可以参见 [spell.py](http://www.norvig.com/spell.py).

tests1 = { 'access': 'acess', 'accessing': 'accesing', 'accommodation':

    'accomodation acommodation acomodation', 'account': 'acount', ...}

tests2 = {'forbidden': 'forbiden', 'decisions': 'deciscions descisions',

    'supposedly': 'supposidly', 'embellishing': 'embelishing', ...}  
  
def spelltest(tests, bias=None, verbose=False):

    import time

    n, bad, unknown, start = 0, 0, 0, time.clock()

    if bias:

        for target in tests: NWORDS[target] += bias

    for target,wrongs in tests.items():

        for wrong in wrongs.split():

            n += 1

            w = correct(wrong)

            if w!=target:

                bad += 1

                unknown += (target not in NWORDS)

                if verbose:

                    print '%r => %r (%d); expected %r (%d)' % (

                        wrong, w, NWORDS[w], target, NWORDS[target])

    return dict(bad=bad, n=n, bias=bias, pct=int(100. - 100.\*bad/n),

                unknown=unknown, secs=int(time.clock()-start) )  
  
print spelltest(tests1)  
print spelltest(tests2) ## only do this after everything is debugged

这个程序给出了下面的输出:

{'bad': 68, 'bias': None, 'unknown': 15, 'secs': 16, 'pct': 74, 'n': 270}  
{'bad': 130, 'bias': None, 'unknown': 43, 'secs': 26, 'pct': 67, 'n': 400}

在270个测试样本上 270 , 我大约能在13秒内得到 74% 的正确率 (每秒17个正确词), 在测试集上, 我得到 67% 正确率 (每秒 15 个).

更新: 在这篇文章的原来版本中, 我把结果错误的报告高了. 原因是程序中一个小bug. 虽然这个 bug 很不起眼, 但我实际上应该能够避免. 我为对阅读我老版本的这篇文章的读者造成感到抱歉. 在 spelltest 源程序的第四行, 我忽略了if bias:  并且把 bias 默认值赋值为0. 我原来想: 如果 bias 是0 , NWORDS[target] += bias 这个语句就不起作用. 而实际上, 虽然这个语句没有改变 NWORDS[target] 的值, 这个却让 (target in NWORDS) 为真. 这样的话, spelltest 就会把训练集合中那些不认识的正确拼写的单词都当成认识来处理了, 程序就会"作弊". 我很喜欢 defaultdict 的简洁, 所以在程序中使用了它, 如果使用 dicts 就不会有这个问题了.2

结论: 我达到了简洁, 快速开发和运行速度这三个目标, 不过准确率不算太好.

## 将来工作

怎样才能做到更好结果呢? 让我们回过头来看看概率模型中的三个因素:  (1) P(*c*); (2) P(*w*|*c*); and (3) argmax*c*. 我们通过程序给出错误答案的那些例子入手, 看看这三个因素外, 我们还忽略了什么.

1、P(*c*), 语言模型. 在语言模型中, 有两种问题会造成最后的错误识别. 其中最严重的一个因素就是 *未知单词*. 在训练集合中, 一共有15个未知单词, 它们大约占了5%; 在测试集合中, 有43个未知词, 它们占了11%. 当把 spelltest 的调用参数 verbose 设置为 True 的时候: 我们可以看到下面的输出:

correct('economtric') => 'economic' (121); expected 'econometric' (1)

correct('embaras') => 'embargo' (8); expected 'embarrass' (1)

correct('colate') => 'coat' (173); expected 'collate' (1)

correct('orentated') => 'orentated' (1); expected 'orientated' (1)

correct('unequivocaly') => 'unequivocal' (2); expected 'unequivocally' (1)

correct('generataed') => 'generate' (2); expected 'generated' (1)

correct('guidlines') => 'guideline' (2); expected 'guidelines' (1)

在这个结果中, 我们可以使用看到 correct 函数作用在那些拼错的单词上的结果. (其中 NWORDS 中单词出现次数在括号中),  然后是我们期望的输出以及出现的次数. 这个结果告诉我们, 如果程序根本就不知道 'econometric' 是一个单词, 它也就不可能去把 'economtric' 纠正成 'econometric'. 这个问题可以通过往训练集合中加入更多语料来解决, 不过也有可能引入更多错误. 同时注意到最后四行, 实际上我们的训练集中有正确的单词, 只是形式略有不同. 因此, 我们可以改进一下程序, 比如在动词后面加 '-ed' 或者在名词后面加 '-s' 也是合法的.

第二个可能导致错误的因素是概率: 两个词都出现在我们的字典里面了, 但是恰恰我们选的概率大的那个不是用户想要的. 不过我要说的是这个问题其实不是最严重的, 也不是独立发生的, 其他原因可能更加严重.

我们可以模拟一下看看如果我们提高语言模型, 最后结果能好多少. 比如说, 我们在训练集上小"作弊"一下. 我们在 spelltest 函数中有一个参数叫做 bias, 实际上就是代表把正确的拼写词多添加几次, 以便提高语言模型中相应的概率. 比如说, 在语料中, 假设正确的词出现的频率多了1次, 或者10次, 或者更多. 如果我们增加 bias 这个参数的值, 可以看到训练集和测试集上的准确率都显著提高了.

|  |  |  |
| --- | --- | --- |
| **Bias** | **训练集.** | **测试集** |
| 0 | 74% | 67% |
| 1 | 74% | 70% |
| 10 | 76% | 73% |
| 100 | 82% | 77% |
| 1000 | 89% | 80% |

在两个集合上我们都能做到大约 80-90%. 这个显示出如果我们有一个好的语言模型, 我们或能达到准确率这个目标. 不过, 这个显得过于乐观了, 因为构建一个更大的语言模型会引入新的词, 从而可能还会引入一些错误结果, 尽管这个地方我们没观察到这个现象

处理未知词还有另外一种办法, 比如说, 假如遇到这个词: "electroencephalographicallz", 比较好的纠正的方法是把最后的 "z" 变成 "y", 因为 '-cally' 是英文中很常见的一个后缀. 虽然 "electroencephalographically" 这个词也不在我们的字典中, 我们也能通过基于音节或者前缀后缀等性质给出拼写建议. 当然, 这种简单前后缀判断的方法比基于构词法的要简单的多.

2、P(*w*|*c*) 是误差模型. 到目前为止, 我们都是用的一个很简陋的模型: 距离越短, 概率越大. 这个也造成了一些问题, 比如下面的例子中, correct 函数返回了编辑距离为 1 的词作为答案, 而正确答案恰恰编辑距离是 2:

correct('reciet') => 'recite' (5); expected 'receipt' (14)

correct('adres') => 'acres' (37); expected 'address' (77)

correct('rember') => 'member' (51); expected 'remember' (162)

correct('juse') => 'just' (768); expected 'juice' (6)

correct('accesing') => 'acceding' (2); expected 'assessing' (1)

举个例子, 程序认为在 'adres' 中把 'd' 变成 'c' 从而得到 'acres' 的优先级比把 d 写成 dd 以及 s 写成 ss 的优先级高, 从而作出了错误的判断. 还有些时候程序在两个编辑距离一样的候选词中选择了错误的一个, 比如:

correct('thay') => 'that' (12513); expected 'they' (4939)

correct('cleark') => 'clear' (234); expected 'clerk' (26)

correct('wer') => 'her' (5285); expected 'were' (4290)

correct('bonas') => 'bones' (263); expected 'bonus' (3)

correct('plesent') => 'present' (330); expected 'pleasant' (97)

这个例子给我们一个同样的教训: 在 'thay' 中, 把 'a' 变成 'e' 的概率比把 'y' 拼成 't' 大. 为了正确的选择 'they', 我们至少要在先验概率上乘以 2.5, 才能使得最后 they 的几率超过 that, 从而选择 they.

显然, 我们可以用一个更好的模型来衡量拼错单词的概率. 比如说, 把一个字母顺手打成两个, 或者把一个元音打成另一个的情况都应该比其他打字错误更加容易发生. 当然, 更好的办法还是从数据入手: 比如说, 找一个拼写错误语料, 然后统计插入; 删除; 交换和变换在给定周围字母情况下的概率. 为了采集到这些概率, 可能我们需要非常大的数据集. 比如说, 如果我们带着观察左右两个字母作为上下文, 看看一个字母替换成另一个的概率, 就一共有 266 种情况, 也就是大约超过 3 亿个情况. 然后每种情况需要平均几个证据作为支撑, 因此我们知道10亿个字母的训练集. 如果为了保证更好的质量, 可能至少100亿个才差不多.  
  
需要注意的是, 语言模型和误差模型之间是有联系的. 我们的程序中假设了编辑距离为 1 的优先于编辑距离为 2 的. 这种误差模型或多或少也使得语言模型的优点难以发挥. 我们之所以没有往语言模型中加入很多不常用的单词, 是因为我们担心添加这些单词后, 他们恰好和我们要更正的词编辑距离是1, 从而那些出现频率更高但是编辑距离为 2 的单词就不可能被选中了. 如果有一个更加好的误差模型, 或许我们就能够放心大胆的添加更多的不常用单词了. 下面就是一个因为添加不常用单词影响结果的例子:

correct('wonted') => 'wonted' (2); expected 'wanted' (214)

correct('planed') => 'planed' (2); expected 'planned' (16)

correct('forth') => 'forth' (83); expected 'fourth' (79)

correct('et') => 'et' (20); expected 'set' (325)

4、枚举所有可能的概率并且选择最大的: argmax*c*. 我们的程序枚举了直到编辑距离为2的所有单词. 在测试集合中, 270个单词中, 只有3个编辑距离大于2, 但是在测试集合中, 400个中却有23个. 他们是:

purple perpul

curtains courtens

minutes muinets

successful sucssuful

hierarchy heiarky

profession preffeson

weighted wagted

inefficient ineffiect

availability avaiblity

thermawear thermawhere

nature natior

dissension desention

unnecessarily unessasarily

disappointing dissapoiting

acquaintances aquantences

thoughts thorts

criticism citisum

immediately imidatly

necessary necasery

necessary nessasary

necessary nessisary

unnecessary unessessay

night nite

minutes muiuets

assessing accesing

necessitates nessisitates

我们可以考虑有限的允许一些编辑距离为3的情况. 比如说, 我们可以只允许在元音旁边插入一个元音, 或者把元音替换, 或者把 c 写成 s 等等. 这些基本上就覆盖了上面所有的情况了.

5、第四种, 也是最好的一种改进方法是改进 correct  函数的接口, 让他可以分析上下文给出决断. 因为很多情况下, 仅仅根据单词本身做决断很难, 这个单词本身就在字典中, 但是在上下文中, 应该被更正为另一个单词. 比如说:

correct('where') => 'where' (123); expected 'were' (452)

correct('latter') => 'latter' (11); expected 'later' (116)

correct('advice') => 'advice' (64); expected 'advise' (20)

如果单看 'where' 这个单词本身, 我们无从知晓说什么情况下该把 correct('where') 返回 'were' , 又在什么情况下返回 'where'. 但是如果我们给 correct 函数的是:'They where going', 这时候 "where" 就应该被更正为 "were".

上下文可以帮助程序从多个候选答案中选出最好的, 比如说:

correct('hown') => 'how' (1316); expected 'shown' (114)

correct('ther') => 'the' (81031); expected 'their' (3956)

correct('quies') => 'quiet' (119); expected 'queries' (1)

correct('natior') => 'nation' (170); expected 'nature' (171)

correct('thear') => 'their' (3956); expected 'there' (4973)

correct('carrers') => 'carriers' (7); expected 'careers' (2)

为什么 'thear' 要被更正为 'there' 而不是 'their' 呢?  只看单词本身, 这个问题不好回答, 不过一旦放句子 'There's no there thear' 中, 答案就立即清楚明了了.

要构建一个同时能处理多个词(词以及上下文)的系统, 我们需要大量的数据. 所幸的是 Google 已经公开发布了最长 5个单词的所有序列[数 据库](http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html), 这个是从上千亿个词的语料数据中收集得到的. 我相信一个能达到 90% 准确率的拼写检查器已经需要考虑上下文以做决定了. 不过, 这个, 咱们改天讨论 :)

6、我们可以通过优化训练数据和测试数据来提高准确率. 我们抓取了大约100万个单词并且假设这些词都是拼写正确的. 但是这个事情并不这么完美, 这些数据集也可能有错. 我们可以尝试这找出这些错并且修正他们. 这个地方, 修正测试集合并不困难. 我留意到至少有三种情况下, 测试集合说我们的程序给出了错误的答案, 而我却认为我们程序的答案比测试集给的答案要好, 比如说: (实际上测试集给的三个答案的拼写都不正确)

correct('aranging') => 'arranging' (20); expected 'arrangeing' (1)

correct('sumarys') => 'summary' (17); expected 'summarys' (1)

correct('aurgument') => 'argument' (33); expected 'auguments' (1)

我们还可以决定英语的变种, 以便训练我们的程序, 比如说下面的三个错误是因为美式英语和英式英语拼发不一样造成的, (我们的训练集两者都有):

correct('humor') => 'humor' (17); expected 'humour' (5)

correct('oranisation') => 'organisation' (8); expected 'organization' (43)

correct('oranised') => 'organised' (11); expected 'organized' (70)

7、最 后的一个改进是让程序运行得更加快一点. 比如说, 我们用编译语言来写, 而不是用解释语言. 我们可以使用查找表, 而不用Python提供的通用的 dict 对象, 我们可以缓存计算结果, 从而避免重复计算, 等等. 一个小建议是: 在做任何速度优化之前, 先弄清楚到底程序的时间花在什么地方了.

## 延伸阅读

* Roger Mitton 写了一个关于拼写检查的[综述](http://www.dcs.bbk.ac.uk/~roger/spellchecking.html).
* Jurafsky 和 Martin 在他们编撰的教材 *[Speech and Language Processing](http://www.cs.colorado.edu/~martin/slp.html)* 中很好的讲述了拼写检察.
* Manning 和 Schutze 在他们编撰的 *[Foundations of Statistical Natural Language Processing](http://nlp.stanford.edu/fsnlp/)* 中很好的讲述了统计语言模型, 但是好像没讲拼写检查. (至少目录中没提).
* [Aspell](http://aspell.net/) 计划含有很多有趣的材料, 比如一些[测试数据](http://aspell.net/test/), 看上去比我用的数据好.

## 订正

我原始的程序一共 20行, 不过 Ivan Peev 指出说我的 string.lowercase, 在某些 locale 和某些版本的 Python中会包含 a-z 以外的更多字母, 因此我加了一个字母表, 我也可以使用 string.ascii\_lowercase.

感谢 Jay Liang 指出一共 54n+25 个编辑距离为 1的词, 而不是 55n+25 个.

感谢Dmitriy Ryaboy 指出 NWORDS[target] += bias bug.

## 其他编程语言实现

我发表这个文章后, 很多人用其他语言实现了. 我的目的是算法而不是 Python. 对于那些比较不同语言的人, 这些其他语言实现可能很有意思:

* Erlang: by [Federico Feroldi](http://www.pixzone.com/blog/223/spell-corrector-aka-google-suggest-in-erlang-first-part/)
* F#: by [Sebastian G](http://cs.hubfs.net/forums/thread/3085.aspx)
* Haskell: by [Grzegorz](http://pithekos.net/brainwave/)
* Perl: by [riffraff](http://www.riffraff.info/2007/5/10/a-spell-corrector-in-perl6-part-1)
* Rebol: by [Cyphre](http://www.rebol.cz/~cyphre/spell.r)
* Ruby: by [Brian Adkins](http://www.ruby-forum.com/topic/104327)
* Scheme: by [Shiro](http://practical-scheme.net/wiliki/wiliki.cgi?Gauche:SpellingCorrection&l=en)
* Scheme: by [Jens Axel](http://scheme.dk/blog/2007/04/writing-spelling-corrector-in-plt.html)

## 其他自然语言翻译

* [日文](http://www.aoky.net/articles/peter_norvig/spell-correct.htm) by Yasushi Aoki
* [俄 文](http://gmdidro.googlepages.com/Ru_HowtoWriteaSpellingCorrector.html) by Petrov Alexander

译注:  
  
1. 这个地方显然作者是为了简化程序, 实际上don't 一般都按照 dont 来处理.   
  
2. 如果程序把训练集合中正确的目标词存放到 NWORDS 中, 就等价于提前知道答案了, 如果这个错误的词编辑距离为 2 之内没有其他正确词, 只有一个这个答案, 程序肯定会选取这个答案. 这个就是所谓的"作弊".