

文本挖掘的分词原理

在做文本挖掘的时候，首先要做的预处理就是分词。英文单词天然有空格隔开容易按照空格分词，但是也有时候需要把多个单词作为一个分词，比如一些名词如“New York”，需要做为一个词看待。而中文由于没有空格，分词就是一个需要专门去解决的问题了。无论是英文还是中文，分词的原理都是类似的，本文就对文本挖掘时的分词原理做一个总结。

1. 分词的基本原理

现代分词都是基于统计的分词，而统计的样本内容来自于一些标准的语料库。假如有一个句子：“小明来到荔湾区”，我们期望语料库统计后分词的结果是：“小明/来到/荔湾/区”，而不是“小明/来到/荔/湾区”。那么如何做到这一点呢？

从统计的角度，我们期望“小明/来到/荔湾/区”这个分词后句子出现的概率要比“小明/来到/荔/湾区”大。如果用数学的语言来说说，如果有n个句子S，它有m种分词选项如下：

$$A_{11}A_{12}\dots A_{1n_1}$$

$$A_{21}A_{22}\dots A_{2n_2}$$

.....

$$A_{m1}A_{m2}\dots A_{mn_m}$$

其中下标*n_i*代表第*i*种分词的词个数。如果我们从中选择了最优的第*r*种分词方法，那么这种分词方法对应的统计分布概率应该最大，即：

$$r = \underbrace{\arg \max_i}_{i} P(A_{i1}, A_{i2}, \dots, A_{in_i})$$

但是我们的概率分布 $P(A_{i1}, A_{i2}, \dots, A_{in_i})$ 并不好求出来，因为它涉及到*n_i*个分词的联合分布。在NLP中，为了简化计算，我们通常使用马尔科夫假设，即每一个分词出现的概率仅仅和前一个分词有关，即：

$$P(A_{ij}|A_{i1}, A_{i2}, \dots, A_{i(j-1)}) = P(A_{ij}|A_{i(j-1)})$$

在前面我们讲MCMC采样时，也用到了相同的假设来简化模型复杂度。使用了马尔科夫假设，则我们的联合分布就好求了，即：

$$P(A_{i1}, A_{i2}, \dots, A_{in_i}) = P(A_{i1})P(A_{i2}|A_{i1})P(A_{i3}|A_{i2})\dots P(A_{in_i}|A_{i(n_i-1)})$$

而通过我们的标准语料库，我们可以近似的计算出所有的分词之间的二元条件概率，比如任意两个词 w_1, w_2 ，它们的条件概率分布可以近似的表示为：

$$P(w_2|w_1) = \frac{P(w_1, w_2)}{P(w_1)} \approx \frac{freq(w_1, w_2)}{freq(w_1)}$$
$$P(w_1|w_2) = \frac{P(w_2, w_1)}{P(w_2)} \approx \frac{freq(w_1, w_2)}{freq(w_2)}$$

其中 $freq(w_1, w_2)$ 表示 w_1, w_2 在语料库中相邻一起出现的次数，而其中 $freq(w_1), freq(w_2)$ 分别表示 w_1, w_2 在语料库中出现的统计次数。

利用语料库建立的统计概率，对于一个新的句子，我们就可以通过计算各种分词方法对应的联合分布概率，找到最大概率对应的分词方法，即为最优分词。

2. N元模型

当然，你会说，只依赖于前一个词太武断了，我们能不能依赖于前两个词呢？即：

$$P(A_{i1}, A_{i2}, \dots, A_{in_i}) = P(A_{i1})P(A_{i2}|A_{i1})P(A_{i3}|A_{i1}, A_{i2})\dots P(A_{in_i}|A_{i(n_i-2)}, A_{i(n_i-1)})$$

这样也是可以的，只不过这样联合分布的计算量就大大增加了。我们一般称只依赖于前一个词的模型为二元模型(Bi-Gram model)，而依赖于前两个词的模型为三元模型。以此类推，我们可以建立四元模型，五元模型，...一直到通用的N元模型。越往后，概率分布的计算复杂度越高。当然算法的原理是类似的。

在实际应用中，N一般都较小，一般都小于4，主要原因是N元模型概率分布的空间复杂度为 $O(|V|^N)$ ，其中 $|V|$ 为语料库大小，而N为模型的元数，当N增大时，复杂度呈指数级的增长。

*N*元模型的分词方法虽然很好，但是在实际中应用也有很多问题，首先，某些生僻词，或者相邻分词联合分布在语料库中没有，概率为0。这种情况我们一般会使用拉普拉斯平滑，即给它一个较小的概率值，这个方法在朴素贝叶斯算法原理小结也有讲到。第二个问题是如果句子长，分词有很多情况，计算量也非常大，这时我们可以用下一节维特比算法来优化算法时间复杂度。

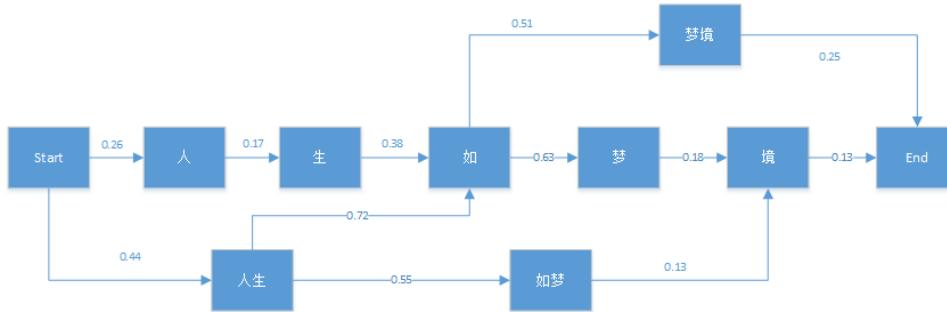
3. 维特比算法与分词

为了简化原理描述，我们本节的讨论都是以二元模型为基础。

对于一个有很多分词可能的长句子，我们当然可以用暴力方法去计算出所有的分词可能的概率，再找出最优分词方法。但是用维特比算法可以大大简化求出最优分词的时间。

大家一般知道维特比算法是用于隐式马尔科夫模型HMM解码算法的，但是它是一个通用的求序列最短路径的方法，不光可以用于HMM，也可以用于其他的序列最短路径算法，比如最优分词。

维特比算法采用的是动态规划来解决这个最优分词问题的，动态规划要求局部路径也是最优路径的一部分，很显然我们的问题是成立的。首先我们看一个简单的分词例子：“人生如梦境”。它的可能分词可以用下面的概率图表示：



图中的箭头为通过统计语料库而得到的对应的各分词条件概率。比如 $P(\text{生}|\text{人})=0.17$ 。有了这个图，维特比算法需要找到从Start到End之间的一条最短路径。对于在End之前的任意一个当前局部节点，我们需要得到到达该节点的最大概率 δ ，和记录到达当前节点满足最大概率的前一节点位置 Ψ 。

我们先用这个例子来观察维特比算法的过程。首先我们初始化有：

$$\delta(\text{人}) = 0.26 \quad \Psi(\text{人}) = \text{Start} \quad \delta(\text{人生}) = 0.44 \quad \Psi(\text{人生}) = \text{Start}$$

对于节点“生”，它只有一个前向节点，因此有：

$$\delta(\text{生}) = \delta(\text{人})P(\text{生}|\text{人}) = 0.0442 \quad \Psi(\text{生}) = \text{人}$$

对于节点“如”，就稍微复杂一点了，因为它有多个前向节点，我们要计算出到“如”概率最大的路径：

$$\delta(\text{如}) = \max\{\delta(\text{生})P(\text{如}|\text{生}), \delta(\text{人生})P(\text{如}|\text{人生})\} = \max\{0.01680, 0.3168\} = 0.3168 \quad \Psi(\text{如}) = \text{人生}$$

类似的方法可以用于其他节点如下：

$$\delta(\text{如梦}) = \delta(\text{人生})P(\text{如梦}|\text{人生}) = 0.242 \quad \Psi(\text{如梦}) = \text{人生}$$

$$\delta(\text{梦}) = \delta(\text{如})P(\text{梦}|\text{如}) = 0.1996 \quad \Psi(\text{梦}) = \text{如}$$

$$\delta(\text{境}) = \max\{\delta(\text{梦})P(\text{境}|\text{梦}), \delta(\text{如梦})P(\text{境}|\text{如梦})\} = \max\{0.0359, 0.0315\} = 0.0359 \quad \Psi(\text{境}) = \text{梦}$$

$$\delta(\text{梦境}) = \delta(\text{梦境})P(\text{梦境}|\text{如}) = 0.1585 \quad \Psi(\text{梦境}) = \text{如}$$

最后我们看看最终节点End：

$$\delta(\text{End}) = \max\{\delta(\text{梦境})P(\text{End}|\text{梦境}), \delta(\text{境})P(\text{End}|\text{境})\} = \max\{0.0396, 0.0047\} = 0.0396 \quad \Psi(\text{End}) = \text{梦境}$$

由于最后的最优解为“梦境”，现在我们开始用 Ψ 反推：

$$\Psi(\text{End}) = \text{梦境} \rightarrow \Psi(\text{梦境}) = \text{如} \rightarrow \Psi(\text{如}) = \text{人生} \rightarrow \Psi(\text{人生}) = \text{start}$$

从而最终的分词结果为“人生/如/梦境”。是不是很简单呢。

由于维特比算法我会在后面讲隐式马尔科夫模型HMM解码算法时详细解释，这里就不归纳了。

4. 常用分词工具

对于文本挖掘中需要的分词功能，一般我们会用现有的工具。简单的英文分词不需要任何工具，通过空格和标点符号就可以分词了，而进一步的英文分词推荐使用nltk。对于中文分词，则推荐用结巴分词（jieba）。这些工具使用都很简单。你的分词没有特别的需求直接使用这些分词工具就可以了。

5. 结语

分词是文本挖掘的预处理的重要一步，分词完成后，我们可以继续做一些其他的特征工程，比如向量化（vectorize），TF-IDF以及Hash trick，这些我们后面再讲。

文本挖掘预处理之向量化与Hash Trick

在文本挖掘的分词原理中，我们讲到了文本挖掘的预处理的关键一步：“分词”，而在做了分词后，如果我们是做文本分类聚类，则后面关键的特征预处理步骤有向量化或向量化的特例Hash Trick，本文我们就对向量化和特例Hash Trick预处理方法做一个总结。

1. 词袋模型

在讲向量化与Hash Trick之前，我们先说说词袋模型(Bag of Words,简称BoW)。词袋模型假设我们不考虑文本中词与词之间的上下文关系，仅仅只考虑所有词的权重。而权重与词在文本中出现的频率有关。

词袋模型首先会进行分词，在分词之后，通过统计每个词在文本中出现的次数，我们就可以得到该文本基于词的特征，如果将各个文本样本的这些词与对应的词频放在一起，就是我们常说的向量化。向量化完毕后一般也会使用TF-IDF进行特征的权重修正，再将特征进行标准化。再进行一些其他的特征工程后，就可以将数据带入机器学习算法进行分类聚类了。

总结下词袋模型的三部曲：分词 (tokenizing)，统计修订词特征值 (counting) 与标准化 (normalizing)。

与词袋模型非常类似的一个模型是词集模型(Set of Words,简称SoW)，和词袋模型唯一的不同是它仅仅考虑词是否在文本中出现，而不考虑词频。也就是一个词在文本中出现1次和多次特征处理是一样的。在大多数时候，我们使用词袋模型，后面的讨论也是以词袋模型为主。

当然，词袋模型有很大的局限性，因为它仅仅考虑了词频，没有考虑上下文的关系，因此会丢失一部分文本的语义。但是大多数时候，如果我们的目的是分类聚类，则词袋模型表现的很好。

2. 词袋模型之向量化

在词袋模型的统计词频这一步，我们会得到该文本中所有词的词频，有了词频，我们就可以用词向量表示这个文本。这里我们举一个例子，例子直接用scikit-learn的CountVectorizer类来完成，这个类可以帮助我们完成文本的词频统计与向量化，代码如下：

```
from sklearn.feature_extraction.text import CountVectorizer
corpus=[ "I come to China to travel",
         "This is a car polupar in China",
         "I love tea and Apple",
         "The work is to write some papers in science"]
print vectorizer.fit_transform(corpus)
```

我们看看对于上面4个文本的处理输出如下：

```
(0, 16)      1
(0, 3)       1
(0, 15)      2
(0, 4)       1
(1, 5)       1
(1, 9)       1
(1, 2)       1
(1, 6)       1
(1, 14)      1
(1, 3)       1
(2, 1)       1
(2, 0)       1
(2, 12)      1
(2, 7)       1
(3, 10)      1
(3, 8)       1
(3, 11)      1
(3, 18)      1
```

```
(3, 17)      1
(3, 13)      1
(3, 5)       1
(3, 6)       1
(3, 15)      1
```

可以看出4个文本的词频已经统计出，在输出中，左边的括号中的第一个数字是文本的序号，第2个数字是词的序号，注意词的序号是基于所有的文档的。第三个数字就是我们的词频。

我们可以进一步看看每个文本的向量特征和各个特征代表的词，代码如下：

```
print vectorizer.fit_transform(corpus).toarray()
print vectorizer.get_feature_names()
```

输出如下：

```
[[0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0]
 [0 0 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0]
 [1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1]]
[u'and', u'apple', u'car', u'china', u'come', u'in', u'is', u'love', u'papers', u'polupar', u'science',
u'some', u'tea', u'the', u'this', u'to', u'travel', u'work', u'write']
```

可以看到我们一共有19个词，所以4个文本都是19维的特征向量。而每一维的向量依次对应了下面的19个词。另外由于词"I"在英文中是停用词，不参加词频的统计。

由于大部分的文本都只会使用词汇表中的很少一部分的词，因此我们的词向量中会有大量的0。也就是说词向量是稀疏的。在实际应用中一般使用稀疏矩阵来存储。

将文本做了词频统计后，我们一般会通过TF-IDF进行词特征值修订，这部分我们后面再讲。

向量化的方法很好用，也很直接，但是在有些场景下很难使用，比如分词后的词汇表非常大，达到100万+，此时如果我们直接使用向量化的方法，将对应的样本对应特征矩阵载入内存，有可能将内存撑爆，在这种情况下我们怎么办呢？第一反应是我们要进行特征的降维，说的没错！而Hash Trick就是非常常用的文本特征降维方法。

3. Hash Trick

在大规模的文本处理中，由于特征的维度对应分词词汇表的大小，所以维度可能非常恐怖，此时需要进行降维，不能直接用我们上一节的向量化方法。而最常用的文本降维方法是Hash Trick。说到Hash，一点也不神秘，学过数据结构的同学都知道。这里的Hash意义也类似。

在Hash Trick里，我们会定义一个特征Hash后对应的哈希表的大小，这个哈希表的维度会远远小于我们的词汇表的特征维度，因此可以看成是降维。具体的方法是，对应任意一个特征名，我们会用Hash函数找到对应哈希表的位置，然后将该特征名对应的词频统计值累加到该哈希表位置。如果用数学语言表示，假如哈希函数 h 使第 i 个特征哈希到位置 j ，即 $h(i) = j$ ，则第 i 个原始特征的词频数值 $\phi(i)$ 将累加到哈希后的第 j 个特征的词频数值 $\bar{\phi}$ 上，即：

$$\bar{\phi}(j) = \sum_{i \in \mathcal{J}; h(i)=j} \phi(i)$$

其中 \mathcal{J} 是原始特征的维度。

但是上面的方法有一个问题，有可能两个原始特征的哈希后位置在一起导致词频累加特征值突然变大，为了解决这个问题，出现了hash Trick的变种signed hash trick，此时除了哈希函数 h ，我们多了一个一个哈希函数：

$$\xi : \mathbb{N} \rightarrow \pm 1$$

此时我们有

$$\bar{\phi}(j) = \sum_{i \in \mathcal{J}; h(i)=j} \xi(i) \phi(i)$$

这样做的好处是，哈希后的特征仍然是一个无偏的估计，不会导致某些哈希位置的值过大。

当然，大家会有疑惑，这种方法来处理特征，哈希后的特征是否能够很好的代表哈希前的特征呢？从实际应用中说，由于文本特征的高稀疏性，这么做是可行的。如果大家对理论上为何这种方法有效，建议参考论文：[Feature hashing for large scale multitask learning](#)，这里就不多说了。

在scikit-learn的HashingVectorizer类中，实现了基于signed hash trick的算法，这里我们就用HashingVectorizer来实践一下Hash Trick，为了简单，我们使用上面的19维词汇表，并哈希降维到6维。当然在实际应用中，19维的数据根本不需要Hash Trick，这里只是做一个演示，代码如下：

```
from sklearn.feature_extraction.text import HashingVectorizer
vectorizer2=HashingVectorizer(n_features = 6, norm = None)
print vectorizer2.fit_transform(corpus)
```

输出如下：

(0, 1)	2.0
(0, 2)	-1.0
(0, 4)	1.0
(0, 5)	-1.0
(1, 0)	1.0
(1, 1)	1.0
(1, 2)	-1.0
(1, 5)	-1.0
(2, 0)	2.0
(2, 5)	-2.0
(3, 0)	0.0
(3, 1)	4.0
(3, 2)	-1.0
(3, 3)	1.0
(3, 5)	-1.0

大家可以看到结果里面有负数，这是因为我们的哈希函数可以哈希到1或者-1导致的。

和PCA类似，Hash Trick降维后的特征我们已经不知道它代表的特征名字和意义。此时我们不能像上一节向量化时候可以知道每一列的意义，所以Hash Trick的解释性不强。

4. 向量化与Hash Trick小结

这里我们对向量化与它的特例Hash Trick做一个总结。在特征预处理的时候，我们什么时候用一般意义的向量化，什么时候用Hash Trick呢？标准也很简单。

一般来说，只要词汇表的特征不至于太大，大到内存不够用，肯定是使用一般意义的向量化比较好。因为向量化的方法解释性很强，我们知道每一维特征对应哪一个词，进而我们还可以使用TF-IDF对各个词特征的权重修改，进一步完善特征的表示。

而Hash Trick用大规模机器学习上，此时我们的词汇量极大，使用向量化方法内存不够用，而使用Hash Trick降维速度很快，降维后的特征仍然可以帮我们完成后续的分类和聚类工作。当然由于分布式计算框架的存在，其实一般我们不会出现内存不够的情况。因此，实际工作中我使用的都是特征向量化。

向量化与Hash Trick就介绍到这里，下一篇我们讨论TF-IDF。

文本挖掘预处理之TF-IDF

在文本挖掘预处理之向量化与Hash Trick中我们讲到在文本挖掘的预处理中，向量化之后一般都伴随着TF-IDF的处理，那么什么是TF-IDF，为什么一般我们要加这一步预处理呢？这里就对TF-IDF的原理做一个总结。

1. 文本向量化特征的不足

在将文本分词并向量化后，我们可以得到词汇表中每个词在各个文本中形成的词向量，比如在文本挖掘预处理之向量化与Hash Trick这篇文章中，我们将下面4个短文本做了词频统计：

```
corpus=[ "I come to China to travel",
        "This is a car polupar in China",
        "I love tea and Apple",
        "The work is to write some papers in science"]
```

不考虑停用词，处理后得到的词向量如下：

```
[[0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 2 1 0 0]
 [0 0 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0]
 [1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 1]]
```

如果我们直接将统计词频后的19维特征做为文本分类的输入，会发现有一些问题。比如第一个文本，我们发现“come”，“China”和“Travel”各出现1次，而“to”出现了两次。似乎看起来这个文本与“to”这个特征更关系紧密。但是实际上“to”是一个非常普遍的词，几乎所有的文本都会用到，因此虽然它的词频为2，但是重要性却比词频为1的“China”和“Travel”要低的多。如果我们的向量化特征仅仅用词频表示就无法反应这一点。因此我们需要进一步的预处理来反应文本的这个特征，而这个预处理就是TF-IDF。

2. TF-IDF概述

TF-IDF是Term Frequency - Inverse Document Frequency的缩写，即“词频-逆文本频率”。它由两部分组成，TF和IDF。

前面的TF也就是我们前面说到的词频，我们之前做的向量化也就是做了文本中各个词的出现频率统计，并作为文本特征，这个很好理解。关键是后面的这个IDF，即“逆文本频率”如何理解。在上一节中，我们讲到几乎所有文本都会出现的“to”其词频虽然高，但是重要性却应该比词频低的“China”和“Travel”要低。我们的IDF就是来帮助我们来反应这个词的重要性，进而修正仅仅用词频表示的词特征值。

概括来讲，IDF反应了一个词在所有文本中出现的频率，如果一个词在很多的文本中出现，那么它的IDF值应该低，比如上文中的“to”。而反过来如果一个词在比较少的文本中出现，那么它的IDF值应该高。比如一些专业的名词如“Machine Learning”。这样的词IDF值应该高。一个极端的情况，如果一个词在所有的文本中都出现，那么它的IDF值应该为0。

上面是从定性上说明的IDF的作用，那么如何对一个词的IDF进行定量分析呢？这里直接给出一个词 x 的IDF的基本公式如下：

$$IDF(x) = \log \frac{N}{N(x)}$$

其中， N 代表语料库中文本的总数，而 $N(x)$ 代表语料库中包含词 x 的文本总数。为什么IDF的基本公式应该是是上面这样的而不是像 $N/N(x)$ 这样的形式呢？这就涉及到信息论相关的一些知识了。感兴趣的朋友建议阅读吴军博士的《数学之美》第11章。

上面的IDF公式已经可以使用了，但是在一些特殊的情况会有一些小问题，比如某一个生僻词在语料库中没有，这样我们的分母为0，IDF没有意义了。所以常用的IDF我们需要做一些平滑，使语料库中没有出现的词也可以得到一个合适的IDF值。平滑的方法有很多种，最常见的IDF平滑后的公式之一为：

$$IDF(x) = \log \frac{N + 1}{N(x) + 1} + 1$$

有了IDF的定义，我们就可以计算某一个词的TF-IDF值了：

$$TF - IDF(x) = TF(x) * IDF(x)$$

其中 $TF(x)$ 指词 x 在当前文本中的词频。

3. 用scikit-learn进行TF-IDF预处理

在scikit-learn中，有两种方法进行TF-IDF的预处理。

第一种方法是在用CountVectorizer类向量化之后再调用TfidfTransformer类进行预处理。第二种方法是直接用TfidfVectorizer完成向量化与TF-IDF预处理。

首先我们来看第一种方法，CountVectorizer+TfidfTransformer的组合，代码如下：

```
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer

corpus=["I come to China to travel",
        "This is a car polupar in China",
        "I love tea and Apple",
        "The work is to write some papers in science"]

vectorizer=CountVectorizer()

transformer = TfidfTransformer()
tfidf = transformer.fit_transform(vectorizer.fit_transform(corpus))
print tfidf
```

输出的各个文本各个词的TF-IDF值如下：

(0, 4)	0.442462137895
(0, 15)	0.697684463384
(0, 3)	0.348842231692
(0, 16)	0.442462137895
(1, 3)	0.357455043342
(1, 14)	0.453386397373
(1, 6)	0.357455043342
(1, 2)	0.453386397373
(1, 9)	0.453386397373
(1, 5)	0.357455043342
(2, 7)	0.5
(2, 12)	0.5
(2, 0)	0.5
(2, 1)	0.5
(3, 15)	0.281131628441
(3, 6)	0.281131628441
(3, 5)	0.281131628441
(3, 13)	0.356579823338
(3, 17)	0.356579823338
(3, 18)	0.356579823338
(3, 11)	0.356579823338
(3, 8)	0.356579823338
(3, 10)	0.356579823338

现在我们用TfidfVectorizer一步到位，代码如下：

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf2 = TfidfVectorizer()
re = tfidf2.fit_transform(corpus)
print re
```

输出的各个文本各个词的TF-IDF值和第一种的输出完全相同。大家可以自己去验证一下。

由于第二种方法比较的简洁，因此在实际应用中推荐使用，一步到位完成向量化，TF-IDF与标准化。

4. TF-IDF小结

TF-IDF是非常常用的文本挖掘预处理基本步骤，但是如果预处理中使用了Hash Trick，则一般就无法使用TF-IDF了，因为Hash Trick后我们已经无法得到哈希后的各特征的IDF的值。使用了TF-IDF并标准化以后，我们就可以使用各个文本的词特征向量作为文本的特征，进行分类或者聚类分析。

当然TF-IDF不光可以用于文本挖掘，在信息检索等很多领域都有使用。因此值得好好的理解这个方法的思想。

中文文本挖掘预处理流程总结

在对文本做数据分析时，我们一大半的时间都会花在文本预处理上，而中文和英文的预处理流程稍有不同，本文就对中文文本挖掘的预处理流程做一个总结。

1. 中文文本挖掘预处理特点

首先我们看看中文文本挖掘预处理和英文文本挖掘预处理相比的一些特殊点。

首先，中文文本是没有像英文的单词空格那样隔开的，因此不能直接像英文一样可以直接用最简单的空格和标点符号完成分词。所以一般我们需要用分词算法来完成分词，在[文本挖掘的分词原理](#)中，我们已经讲到了中文的分词原理，这里就不多说。

第二，中文的编码不是utf8，而是unicode。这样会导致在分词的时候，和英文相比，我们要处理编码的问题。

这两点构成了中文分词相比英文分词的一些不同点，后面我们也会重点讲述这部分的处理。当然，英文分词也有自己的烦恼，这个我们在以后再讲。了解了中文预处理的一些特点后，我们就言归正传，通过实践总结下中文文本挖掘预处理流程。

2. 中文文本挖掘预处理一：数据收集

在文本挖掘之前，我们需要得到文本数据，文本数据的获取方法一般有两种：使用别人做好的语料库和自己用爬虫去在网上爬自己的语料数据。

对于第一种方法，常用的文本语料库在网上有很多，如果大家只是学习，则可以直接下载下来使用，但如果是某些特殊主题的语料库，比如“机器学习”相关的语料库，则这种方法行不通，需要我们自己用第二种方法去获取。

对于第二种使用爬虫的方法，开源工具有很多，通用的爬虫我一般使用[beautifulsoup](#)。但是我们我们需要某些特殊的语料数据，比如上面提到的“机器学习”相关的语料库，则需要用主题爬虫（也叫聚焦爬虫）来完成。这个我一般使用[ache](#)，[ache](#)允许我们用关键字或者一个分类算法来过滤出我们需要的主题语料，比较强大。

3. 中文文本挖掘预处理二：除去数据中非文本部分

这一步主要是针对我们用爬虫收集的语料数据，由于爬下来的内容中有很多html的一些标签，需要去掉。少量的非文本内容的可以直接用Python的正则表达式(re)删除，复杂的则可以用[beautifulsoup](#)来去除。去除掉这些非文本的内容后，我们就可以进行真正的文本预处理了。

4. 中文文本挖掘预处理三：处理中文编码问题

由于Python2不支持unicode的处理，因此我们使用Python2做中文文本预处理时需要遵循的原则是，存储数据都用utf8，读出来进行中文相关处理时，使用GBK之类的中文编码，在下面一节的分词时，我们再用例子说明这个问题。

5. 中文文本挖掘预处理四：中文分词

常用的中文分词软件有很多，个人比较推荐结巴分词。安装也很简单，比如基于Python的，用“`pip install jieba`”就可以完成。下面我们就用例子来看看如何中文分词。

首先我们准备了两段文本，这两段文本在两个文件中。两段文本的内容分别是nlp_test0.txt和nlp_test2.txt：

沙瑞金赞叹易学习的胸怀，是金山的百姓有福，可是这件事对李达康的触动很大。易学习又回忆起他们三人分开的前一晚，大家一起喝酒话别，易学习被降职到道口县当县长，王大路下海经商，李达康连连赔礼道歉，觉得对不起大家，他最对不起的是王大路，就和易学习一起给王大路凑了5万块钱，王大路自己东挪西撮了5万块，开始下海经商。没想到后来王大路竟然做得风生水起。沙瑞金觉得他们三人，在困难时期还能以沫相助，很不容易。

沙瑞金向毛娅打听他们家在京州的别墅，毛娅笑着说，王大路事业有成之后，要给欧阳菁和她公司的股权，她们没有要，王大路就在京州帝豪园买了三套别墅，可是李达康和易学习都不要，这些房子都在王大路的名下，欧阳菁好像去住过，毛娅不想去，她觉得房子太大很浪费，自己住得就很踏实。

我们先讲文本从第一个文件中读取，并使用中文GBK编码，再调用结巴分词，最后把分词结果用uft8格式存在另一个文本nlp_test1.txt

中。代码如下：



```
# -*- coding: utf-8 -*-

import jieba

with open('./nlp_test0.txt') as f:
    document = f.read()

    document_decode = document.decode('GBK')
    document_cut = jieba.cut(document_decode)
    #print ' '.join(jieba_cut) //如果打印结果，则分词效果消失，后面的result无法显示
    result = ' '.join(document_cut)
    result = result.encode('utf-8')
    with open('./nlp_test1.txt', 'w') as f2:
        f2.write(result)
f.close()
f2.close()
```



输出的文本内容如下：

沙瑞金赞叹易学习的胸怀，是金山的百姓有福，可是这件事对李达康的触动很大。易学习又回忆起他们三人分开的前一晚，大家一起喝酒话别，易学习被降职到道口县当县长，王大路下海经商，李达康连连赔礼道歉，觉得对不起大家，他最对不起的是王大路，就和易学习一起给王大路凑了5万块钱，王大路自己东挪西撮了5万块，开始下海经商。没想到后来王大路竟然做得风生水起。沙瑞金觉得他们三人，在困难时期还能以沫相助，很不容易。

可以发现对于一些人名和地名，jieba处理的不好，不过我们可以帮jieba加入词汇如下：

```
jieba.suggest_freq('沙瑞金', True)
jieba.suggest_freq('易学习', True)
jieba.suggest_freq('王大路', True)
jieba.suggest_freq('京州', True)
```

现在我们再来进行读文件，编码，分词，编码和写文件，代码如下：

```
with open('./nlp_test0.txt') as f:
    document = f.read()

    document_decode = document.decode('GBK')
    document_cut = jieba.cut(document_decode)
    #print ' '.join(jieba_cut)
    result = ' '.join(document_cut)
    result = result.encode('utf-8')
    with open('./nlp_test1.txt', 'w') as f2:
        f2.write(result)
f.close()
f2.close()
```



输出的文本内容如下：

沙瑞金赞叹易学习的胸怀，是金山的百姓有福，可是这件事对李达康的触动很大。易学习又回忆起他们三人分开的前一晚，大家一起喝酒话别，易学习被降职到道口县当县长，王大路下海经商，李达康连连赔礼道歉，觉得对不起大家，他最对不起的是王大路，就和易学习一起给王大路凑了5万块钱，王大路自己东挪西撮了5万块，开始下海经商。没想到后来王大路竟然做得风生水起。沙瑞金觉得他们三人，在困难时期还能以沫相助，很不容易。

基本已经可以满足要求。同样的方法我们对第二段文本nlp_test2.txt进行分词和写入文件nlp_test3.txt。

```
with open('./nlp_test2.txt') as f:
    document2 = f.read()

    document2_decode = document2.decode('GBK')
    document2_cut = jieba.cut(document2_decode)
    #print ' '.join(jieba_cut)
    result = ' '.join(document2_cut)
```

```
result = result.encode('utf-8')
with open('./nlp_test3.txt', 'w') as f2:
    f2.write(result)
f.close()
f2.close()
```



输出的文本内容如下：

沙瑞金向毛娅打听他们家在京州的别墅，毛娅笑着说，王大路事业有成之后，要给欧阳菁和她公司的股权，她们没有要，王大路就在京州帝豪园买了三套别墅，可是李达康和易学习都不要，这些房子都在王大路的名下，欧阳菁好像去住过，毛娅不想去，她觉得房子太大很浪费，自己家住得就很踏实。

可见分词效果还不错。

6. 中文文本挖掘预处理五：引入停用词

在上面我们解析的文本中有很多无效的词，比如“着”，“和”，还有一些标点符号，这些我们不想在文本分析的时候引入，因此需要去掉，这些词就是停用词。常用的中文停用词表是1208个，[下载地址在这](#)。当然也有其他版本的停用词表，不过这个1208词版是我常用的。

在我们用scikit-learn做特征处理的时候，可以通过参数stop_words来引入一个数组作为停用词表。

现在我们将停用词表从文件读出，并切分成一个数组备用：

```
#从文件导入停用词表
stpwrddpath = "stop_words.txt"
stpwrddic = open(stpwrddpath, 'rb')
stpwrddcontent = stpwrddic.read()
#将停用词表转换为list
stpwrddlst = stpwrddcontent.splitlines()
stpwrddic.close()
```



7. 中文文本挖掘预处理六：特征处理

现在我们就可以用scikit-learn来对我们的文本特征进行处理了，在[文本挖掘预处理之向量化与Hash Trick](#)中，我们讲到了两种特征处理的方法，向量化与Hash Trick。而向量化是最常用的方法，因为它可以接着进行TF-IDF的特征处理。在[文本挖掘预处理之TF-IDF](#)中，我们也讲到了TF-IDF特征处理的方法。这里我们就用scikit-learn的TfidfVectorizer类来进行TF-IDF特征处理。

TfidfVectorizer类可以帮助我们完成向量化，TF-IDF和标准化三步。当然，还可以帮我们处理停用词。

现在我们把上面分词好的文本载入内存：

```
with open('./nlp_test1.txt') as f3:
    res1 = f3.read()
print res1
with open('./nlp_test3.txt') as f4:
    res2 = f4.read()
print res2
```



这里的输出还是我们上面分完词的文本。现在我们可以进行向量化，TF-IDF和标准化三步处理了。注意，这里我们引入了我们上面的停用词表。

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [res1, res2]
vector = TfidfVectorizer(stop_words=stpwrddlst)
tfidf = vector.fit_transform(corpus)
print tfidf
```

部分输出如下：

```
(0, 44)      0.154467434933
(0, 59)      0.108549295069
(0, 39)      0.308934869866
```

```
(0, 53)      0.108549295069
...
(1, 27)      0.139891059658
(1, 47)      0.139891059658
(1, 30)      0.139891059658
(1, 60)      0.139891059658
```

我们再来看看每次词和TF-IDF的对应关系：

```
wordlist = vector.get_feature_names() #获取词袋模型中的所有词
# tf-idf矩阵 元素a[i][j]表示j词在i类文本中的tf-idf权重
weightlist = tfidf.toarray()
#打印每类文本的tf-idf词语权重，第一个for遍历所有文本，第二个for便利某一类文本下的词语权重
for i in range(len(weightlist)):
    print "-----第",i,"段文本的词语tf-idf权重-----"
    for j in range(len(wordlist)):
        print wordlist[j],weightlist[i][j]
```

部分输出如下：

-----第 0 段文本的词语tf-idf权重-----

一起 0.217098590137
万块 0.217098590137
三人 0.217098590137
三套 0.0
下海经商 0.217098590137

....

-----第 1 段文本的词语tf-idf权重-----

....
李达康 0.0995336411066
欧阳 0.279782119316
毛娅 0.419673178975
沙瑞金 0.0995336411066
没想到 0.0
没有 0.139891059658
浪费 0.139891059658
王大路 0.29860092332

....

8. 中文文本挖掘预处理七：建立分析模型

有了每段文本的TF-IDF的特征向量，我们就可以利用这些数据建立分类模型，或者聚类模型了，或者进行主题模型的分析。比如我们上面的两段文本，就可以是两个训练样本了。此时的分类聚类模型和之前讲的非自然语言处理的数据分析没有什么两样。因此对应的算法都可以直接使用。而主题模型是自然语言处理比较特殊的一块，这个我们后面再单独讲。

9.中文文本挖掘预处理总结

上面我们对中文文本挖掘预处理的过程做了一个总结，希望可以帮助到大家。需要注意的是这个流程主要针对一些常用的文本挖掘，并使用了词袋模型，对于某些自然语言处理的需求则流程需要修改。比如我们涉及到词上下文关系的一些需求，此时不能使用词袋模型。而有时候我们对于特征的处理有自己的特殊需求，因此这个流程仅供自然语言处理入门者参考。

下一篇我们来总结英文文本挖掘预处理流程，尽情期待。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

英文文本挖掘预处理流程总结

在中文文本挖掘预处理流程总结中，我们总结了中文文本挖掘的预处理流程，这里我们再对英文文本挖掘的预处理流程做一个总结。

1. 英文文本挖掘预处理特点

英文文本的预处理方法和中文的有部分区别。首先，英文文本挖掘预处理一般可以不做分词（特殊需求除外），而中文预处理分词是必不可少的一步。第二点，大部分英文文本都是uft-8的编码，这样在大多数时候处理的时候不用考虑编码转换的问题，而中文文本处理必须要处理unicode的编码问题。这两部分我们在中文文本挖掘预处理里已经讲了。

而英文文本的预处理也有自己特殊的地方，第三点就是拼写问题，很多时候，我们的预处理要包括拼写检查，比如“Helo World”这样的错误，我们不能在分析的时候讲错纠错，所以需要在预处理前加以纠正。第四点就是词干提取(stemming)和词形还原(lemmatization)。这个东西主要是英文有单数，复数和各种时态，导致一个词会有不同的形式。比如“countries”和“country”，“wolf”和“wolves”，我们期望是有一个词。

后面的预处理中，我们会重点讲述第三点和第四点的处理。

2. 英文文本挖掘预处理一：数据收集

这部分英文和中文类似。获取方法一般有两种：使用别人做好的语料库和自己用爬虫去在网上去爬自己的语料数据。

对于第一种方法，常用的文本语料库在网上有很多，如果大家只是学习，则可以直接下载下来使用，但如果是某些特殊主题的语料库，比如“deep learning”相关的语料库，则这种方法行不通，需要我们自己用第二种方法去获取。

对于第二种使用爬虫的方法，开源工具有很多，通用的爬虫我一般使用beautifulsoup。但是我们需要某些特殊的语料数据，比如上面提到的“deep learning”相关的语料库，则需要用主题爬虫（也叫聚焦爬虫）来完成。这个我一般使用ache。ache允许我们用关键字或者一个分类算法模型来过滤出我们需要的主题语料，比较强大。

3. 英文文本挖掘预处理二：除去数据中非文本部分

这一步主要是针对我们用爬虫收集的语料数据，由于爬下来的内容中有很多html的一些标签，需要去掉。少量的非文本内容的可以直接用Python的正则表达式(re)删除，复杂的则可以用beautifulsoup来去除。另外还有一些特殊的非英文字符(non-alpha)，也可以用Python的正则表达式(re)删除。

4. 英文文本挖掘预处理三：拼写检查更正

由于英文文本中可能有拼写错误，因此一般需要进行拼写检查。如果确信我们分析的文本没有拼写问题，可以略去此步。

拼写检查，我们一般用pyenchant类库完成。pyenchant的安装很简单：“pip install pyenchant”即可。

对于一段文本，我们可以用下面的方式去找出拼写错误：

```
from enchant.checker import SpellChecker
chkr = SpellChecker("en_US")
chkr.set_text("Many peope likee to watch In the Name of People.")
for err in chkr:
    print "ERROR:", err.word
```

输出是：

```
ERROR: peope
ERROR: likee
```

找出错误后，我们可以自己来决定是否要改正。当然，我们也可以用pyenchant中的wxSpellCheckerDialog类来用对话框的形式来交互决定是忽略，改正还是全部改正文本中的错误拼写。大家感兴趣的话可以去研究pyenchant的官方文档。

5. 英文文本挖掘预处理四：词干提取(stemming)和词形还原(lemmatization)

词干提取(stemming)和词型还原(lemmatization)是英文文本预处理的特色。两者其实有共同点，即都是要找到词的原始形式。只不过词干提取(stemming)会更加激进一点，它在寻找词干的时候可以会得到不是词的词干。比

如"imaging"的词干可能得到的是"imag"，并不是一个词。而词形还原则保守一些，它一般只对能够还原成一个正确的词的词进行处理。个人比较喜欢使用词型还原而不是词干提取。

在实际应用中，一般使用nltk来进行词干提取和词型还原。安装nltk也很简单，"pip install nltk"即可。只不过我们一般需要下载nltk的语料库，可以用下面的代码完成，nltk会弹出对话框选择要下载的内容。选择下载语料库就可以了。

```
import nltk  
nltk.download()
```

在nltk中，做词干提取的方法有PorterStemmer，LancasterStemmer和SnowballStemmer。个人推荐使用SnowballStemmer。这个类可以处理很多种语言，当然，除了中文。

```
from nltk.stem import SnowballStemmer  
stemmer = SnowballStemmer("english") # Choose a language  
stemmer.stem("countries") # Stem a word
```

输出是"country"，这个词干并不是一个词。

而如果是做词型还原，则一般可以使用WordNetLemmatizer类，即wordnet词形还原方法。

```
from nltk.stem import WordNetLemmatizer  
wnl = WordNetLemmatizer()  
print(wnl.lemmatize('countries'))
```

输出是"country"，比较符合需求。

在实际的英文文本挖掘预处理的时候，建议使用基于wordnet的词形还原就可以了。

在这里有个词干提取和词型还原的demo，如果是这块的新手可以去看看，上手很合适。

6. 英文文本挖掘预处理五：转化为小写

由于英文单词有大小写之分，我们期望统计时像"Home"和"home"是一个词。因此一般需要将所有的词都转化为小写。这个直接用python的API就可以搞定。

7. 英文文本挖掘预处理六：引入停用词

在英文文本中有很多无效的词，比如"a"，"to"，一些短词，还有一些标点符号，这些我们不想在文本分析的时候引入，因此需要去掉，这些词就是停用词。个人常用的英文停用词表[下载地址在这](#)。当然也有其他版本的停用词表，不过这个版本是我常用的。

在我们用scikit-learn做特征处理的时候，可以通过参数stop_words来引入一个数组作为停用词表。这个方法和前文讲中文停用词的方法相同，这里就不写出代码，大家参考前文即可。

8. 英文文本挖掘预处理七：特征处理

现在我们就可以用scikit-learn来对我们的文本特征进行处理了，在[文本挖掘预处理之向量化与Hash Trick](#)中，我们讲到了两种特征处理的方法，向量化与Hash Trick。而向量化是最常用的方法，因为它可以接着进行TF-IDF的特征处理。在[文本挖掘预处理之TF-IDF](#)中，我们也讲到了TF-IDF特征处理的方法。

TfidfVectorizer类可以帮助我们完成向量化，TF-IDF和标准化三步。当然，还可以帮我们处理停用词。这部分工作和中文的特征处理也是完全相同的，大家参考前文即可。

9. 英文文本挖掘预处理八：建立分析模型

有了每段文本的TF-IDF的特征向量，我们就可以利用这些数据建立分类模型，或者聚类模型了，或者进行主题模型的分析。此时的分类聚类模型和之前讲的非自然语言处理的数据分析没有什么两样。因此对应的算法都可以直接使用。而主题模型是自然语言处理比较特殊的一块，这个我们后面再单独讲。

10. 英文文本挖掘预处理总结

上面我们对英文文本挖掘预处理的过程做了一个总结，希望可以帮助到大家。需要注意的是这个流程主要针对一些常用的文本挖掘，并使用了词袋模型，对于某些自然语言处理的需求则流程需要修改。比如有时候需要做词性标注，而有时候我们也需要英文分词，比如得到"New York"而不是"New"和"York"，因此这个流程仅供自然语言处理入门者参考，我们可以根据我们的数据分析目的选择合适的预处理方法。

文本主题模型之潜在语义索引(LSI)

在文本挖掘中，主题模型是比较特殊的一块，它的思想不同于我们常用的机器学习算法，因此这里我们需要专门来总结文本主题模型的算法。本文关注于潜在语义索引算法(LSI)的原理。

1. 文本主题模型的问题特点

在数据分析中，我们经常会进行非监督学习的聚类算法，它可以对我们的特征数据进行非监督的聚类。而主题模型也是非监督的算法，目的是得到文本按照主题的概率分布。从这个方面来说，主题模型和普通的聚类算法非常的类似。但是两者其实还是有区别的。

聚类算法关注于从样本特征的相似度方面将数据聚类。比如通过数据样本之间的欧式距离，曼哈顿距离的大小聚类等。而主题模型，顾名思义，就是对文字中隐含主题的一种建模方法。比如从“人民的名义”和“达康书记”这两个词我们很容易发现对应的文本有很大的主题相关度，但是如果通过词特征来聚类的话则很难找出，因为聚类方法不能考虑到到隐含的主题这一块。

那么如何找到隐含的主题呢？这是一个大问题。常用的方法一般都是基于统计学的生成方法。即假设以一定的概率选择了一个主题，然后以一定的概率选择当前主题的词。最后这些词组成了我们当前的文本。所有词的统计概率分布可以从语料库获得，具体如何以“一定的概率选择”，这就是各种具体的主题模型算法的任务了。

当然还有一些不是基于统计的方法，比如我们下面讲到的LSI。

2. 潜在语义索引(LSI)概述

潜在语义索引(Latent Semantic Indexing,以下简称LSI)，有的文章也叫Latent Semantic Analysis (LSA)。其实是一个东西，后面我们统称LSI，它是一种简单实用的主题模型。LSI是基于奇异值分解 (SVD) 的方法来得到文本的主题的。而SVD及其应用我们在前面的文章也多次讲到，比如：奇异值分解(SVD)原理与在降维中的应用和矩阵分解在协同过滤推荐算法中的应用。如果大家对SVD还不熟悉，建议复习奇异值分解(SVD)原理与在降维中的应用后再读下面的内容。

这里我们简要回顾下SVD：对于一个 $m \times n$ 的矩阵 A ，可以分解为下面三个矩阵：

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T$$

有时为了降低矩阵的维度到 k ，SVD的分解可以近似的写为：

$$A_{m \times n} \approx U_{m \times k} \Sigma_{k \times k} V_{k \times n}^T$$

如果把上式用到我们的主题模型，则SVD可以这样解释：我们输入的有 m 个文本，每个文本有 n 个词。而 A_{ij} 则对应第*i*个文本的第*j*个词的特征值，这里最常用的是基于预处理后的标准化TF-IDF值。 k 是我们假设的主题数，一般要比文本数少。SVD分解后， U_{il} 对应第*i*个文本和第*l*个主题的相关度。 V_{jm} 对应第*j*个词和第*m*个词义的相关度。 Σ_{lm} 对应第*l*个主题和第*m*个词义的相关度。

也可以反过来解释：我们输入的有 m 个词，对应 n 个文本。而 A_{ij} 则对应第*i*个词档的第*j*个文本的特征值，这里最常用的是基于预处理后的标准化TF-IDF值。 k 是我们假设的主题数，一般要比文本数少。SVD分解后， U_{il} 对应第*i*个词和第*l*个词义的相关度。 V_{jm} 对应第*j*个文本和第*m*个主题的相关度。 Σ_{lm} 对应第*l*个词义和第*m*个主题的相关度。

这样我们通过一次SVD，就可以得到文档和主题的相关度，词和词义的相关度以及词义和主题的相关度。

3. LSI简单实例

这里举一个简单的LSI实例，假设我们有下面这个有10个词三个文本的词频TF对应矩阵如下：

$$\begin{array}{c}
 \text{Terms} \\
 \downarrow \\
 \begin{array}{l}
 \text{arrived} \\
 \text{damaged} \\
 \text{delivery} \\
 \text{fire} \\
 \text{gold} \\
 \text{in} \\
 \text{of} \\
 \text{shipment} \\
 \text{silver} \\
 \text{truck}
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \text{d1} \quad \text{d2} \quad \text{d3} \quad \text{q} \\
 \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
 \mathbf{A} = \left[\begin{array}{ccc|c}
 1 & 1 & 1 & 0 \\
 0 & 1 & 1 & 0 \\
 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 \\
 1 & 0 & 0 & 0 \\
 1 & 0 & 1 & 1 \\
 1 & 1 & 1 & 0 \\
 1 & 1 & 1 & 0 \\
 1 & 0 & 1 & 0 \\
 0 & 2 & 0 & 1 \\
 0 & 1 & 1 & 1
 \end{array} \right] \quad \mathbf{q} = \left[\begin{array}{c}
 0 \\
 0 \\
 0 \\
 0 \\
 0 \\
 1 \\
 0 \\
 0 \\
 0 \\
 1 \\
 1
 \end{array} \right]
 \end{array}$$

这里我们没有使用预处理，也没有使用TF-IDF，在实际应用中最好使用预处理后的TF-IDF值矩阵作为输入。

我们假定对应的主题数为2，则通过SVD降维后得到的三矩阵为：

$$\begin{aligned}
 \mathbf{U} \approx \mathbf{U}_k &= \left[\begin{array}{cc}
 -0.4201 & 0.0748 \\
 -0.2995 & -0.2001 \\
 -0.1206 & 0.2749 \\
 -0.1576 & -0.3046 \\
 -0.1206 & 0.2749 \\
 -0.2626 & 0.3794 \\
 -0.4201 & 0.0748 \\
 -0.4201 & 0.0748 \\
 -0.2626 & 0.3794 \\
 -0.3151 & -0.6093 \\
 -0.2995 & -0.2001
 \end{array} \right] & k = 2 \\
 \Sigma \approx \Sigma_k &= \left[\begin{array}{cc}
 4.0989 & 0.0000 \\
 0.0000 & 2.3616
 \end{array} \right] \\
 \mathbf{V} \approx \mathbf{V}_k &= \left[\begin{array}{cc}
 -0.4945 & 0.6492 \\
 -0.6458 & -0.7194 \\
 -0.5817 & 0.2469
 \end{array} \right] & \mathbf{V}^T \approx \mathbf{V}_k^T = \left[\begin{array}{ccc}
 -0.4945 & -0.6458 & -0.5817 \\
 0.6492 & -0.7194 & 0.2469
 \end{array} \right]
 \end{aligned}$$

从矩阵 \mathbf{U}_k 我们可以看到词和词义之间的相关性。而从 \mathbf{V}_k 可以看到3个文本和两个主题的相关性。大家可以看到里面有负数，所以这样得到的相关度比较难解释。

4. LSI用于文本相似度计算

在上面我们通过LSI得到的文本主题矩阵可以用于文本相似度计算。而计算方法一般是通过余弦相似度。比如对于上面的三文档两主题的例子。我们可以计算第一个文本和第二个文本的余弦相似度如下：

$$sim(d1, d2) = \frac{(-0.4945) * (-0.6458) + (0.6492) * (-0.7194)}{\sqrt{(-0.4945)^2 + 0.6492^2} \sqrt{(-0.6458)^2 + (-0.7194)^2}}$$

5. LSI主题模型总结

LSI是最早出现的主题模型了，它的算法原理很简单，一次奇异值分解就可以得到主题模型，同时解决词义的问题，非常漂亮。但是LSI有很多不足，导致它在当前实际的主题模型中已基本不再使用。

主要的问题有：

1) SVD计算非常的耗时，尤其是我们的文本处理，词和文本数都是非常大的，对于这样的高维度矩阵做奇异值分解是非常难的。

2) 主题值的选取对结果的影响非常大，很难选择合适的k值。

3) LSI得到的不是一个概率模型，缺乏统计基础，结果难以直观的解释。

对于问题1)，主题模型非负矩阵分解(NMF)可以解决矩阵分解的速度问题。对于问题2)，这是老大难了，大部分主题模型的主题的个数选取一般都是凭经验的，较新的层次狄利克雷过程(HDP)可以自动选择主题个数。对于问题3)，牛人们整出了pLSI(也叫pLSA)和隐含狄利克雷分布(LDA)这类基于概率分布的主题模型来替代基于矩阵分解的主题模型。

回到LSI本身，对于一些规模较小的问题，如果想快速粗粒度的找出一些主题分布的关系，则LSI是比较好的一个选择，其他时候，如果你需要使用主题模型，推荐使用LDA和HDP。

文本主题模型之非负矩阵分解(NMF)

在文本主题模型之潜在语义索引(LSI)中，我们讲到LSI主题模型使用了奇异值分解，面临着高维度计算量太大的问题。这里我们就介绍另一种基于矩阵分解的主题模型：非负矩阵分解(NMF)，它同样使用了矩阵分解，但是计算量和处理速度则比LSI快，它是怎么做到的呢？

1. 非负矩阵分解(NMF)概述

非负矩阵分解(non-negative matrix factorization，以下简称NMF)是一种非常常用的矩阵分解方法，它可以适用于很多领域，比如图像特征识别，语音识别等，这里我们会主要关注于它在文本主题模型里的运用。

回顾奇异值分解，它会将一个矩阵分解为三个矩阵：

$$A = U\Sigma V^T$$

如果降维到k维，则表达式为：

$$A_{m \times n} \approx U_{m \times k} \Sigma_{k \times k} V_{k \times n}^T$$

但是NMF虽然也是矩阵分解，它却使用了不同的思路，它的目标是期望将矩阵分解为两个矩阵：

$$A_{m \times n} \approx W_{m \times k} H_{k \times n}$$

分解成两个矩阵是不是一定就比SVD省时呢？这里的理论不深究，但是NMF的确比SVD快。不过如果大家读过我写的矩阵分解在协同过滤推荐算法中的应用，就会发现里面的FunkSVD所用的算法思路和NMF基本是一致的，只不过FunkSVD聚焦于推荐算法而已。

那么如何可以找到这样的矩阵呢？这就涉及到NMF的优化思路了。

2. NMF的优化思路

NMF期望找到这样的两个矩阵W, H，使WH的矩阵乘积得到的矩阵对应的每个位置的值和原矩阵A对应位置的值相比误差尽可能的小。用数学的语言表示就是：

$$\underbrace{\arg \min}_{W,H} \frac{1}{2} \sum_{i,j} (A_{ij} - (WH)_{ij})^2$$

如果完全用矩阵表示，则为：

$$\underbrace{\arg \min}_{W,H} \frac{1}{2} \|A - WH\|_{Fro}^2$$

其中， $\|\cdot\|_{Fro}$ 为Frobenius范数。

当然对于这个式子，我们也可以加上L1和L2的正则化项如下：

$$\underbrace{\arg \min}_{W,H} \frac{1}{2} \|A - WH\|_{Fro}^2 + \alpha \rho \|W\|_1 + \alpha \rho \|H\|_1 + \frac{\alpha(1-\rho)}{2} \|W\|_{Fro}^2 + \frac{\alpha(1-\rho)}{2} \|H\|_{Fro}^2$$

其中， α 为L1&L2正则化参数，而 ρ 为L1正则化占总正则化项的比例。 $\|\cdot\|_1$ 为L1范数。

我们要求解的有 $m * k + k * n$ 个参数。参数不少，常用的迭代方法有梯度下降法和拟牛顿法。不过如果我们决定加上了L1正则化的话就不能用梯度下降和拟牛顿法了。此时可以用坐标轴下降法或者最小角回归法来求解。scikit-learn中NMF的库目前是使用坐标轴下降法来求解的，即在迭代时，一次固定 $m * k + k * n - 1$ 个参数，仅仅最优化一个参数。这里对优化求W, H的过程就不再写了，如果大家对坐标轴下降法不熟悉，参看之前写的这一篇Lasso回归算法：坐标轴下降法与最小角回归法小结。

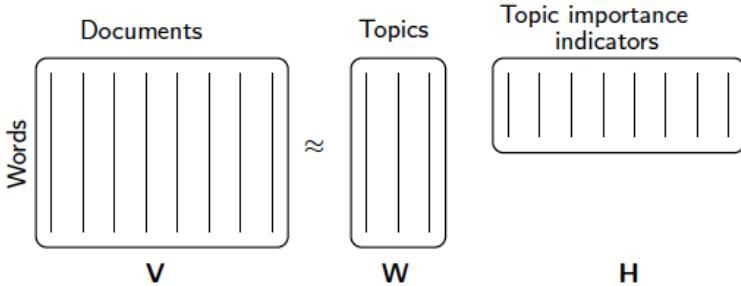
3. NMF 用于文本主题模型

回到我们本文的主题，NMF矩阵分解如何运用到我们的主题模型呢？

此时NMF可以这样解释：我们输入的有m个文本，n个词，而 A_{ij} 对应第i个文本的第j个词的特征值，这里最常用的是基于预处理后的标准化TF-IDF值。k是我们假设的主题数，一般要比文本数少。NMF分解后， W_{ik} 对应第i个文本的和第k个主题的概率相关度，而 H_{kj} 对应第j个词和第k个主题的概率相关度。

当然也可以反过来去解释：我们输入的有m个词，n个文本，而 A_{ij} 对应第i个词的第j个文本的特征值，这里最常用的是基于预处理后的标准化TF-IDF值。k是我们假设的主题数，一般要比文本数少。NMF分解后， W_{ik} 对应第i个词的和第k个主题的概率相关度，而 H_{kj} 对应第j个文本和第k个主题的概率相关度。

注意到这里我们使用的是“概率相关度”，这是因为我们使用的是“非负”的矩阵分解，这样我们的 W, H 矩阵值的大小可以用概率值的角度去看。从而可以得到文本和主题的概率分布关系。第二种解释用一个图来表示如下：



和LSI相比，我们不光得到了文本和主题的关系，还得到了直观的概率解释，同时分解速度也不错。当然NMF由于是两个矩阵，相比LSI的三矩阵，NMF不能解决词和词义的相关度问题。这是一个小小的代价。

4. scikit-learn NMF的使用

在 scikit-learn中，NMF在sklearn.decomposition.NMF包中，它支持L1和L2的正则化，而 W, H 的求解使用坐标轴下降法来实现。

NMF需要注意的参数有：

1) **n_components**：即我们的主题数k，选择k值需要一些对于要分析文本主题大概的先验知识。可以多选择几组k的值进行NMF，然后对结果人为的进行一些验证。

2) **init**：用于帮我们选择 W, H 迭代初值的算法，默认是None，即自动选择值，不使用选择初值的算法。如果我们将收敛速度不满意，才需要关注这个值，从scikit-learn提供的算法中选择一个合适的初值选取算法。

3) **alpha**：即我们第三节中的正则化参数 α ，需要调参。开始建议选择一个比较小的值，如果发现效果不好在调参增大。

4) **l1_ratio**：即我们第三节中的正则化参数 ρ ，L1正则化的比例，仅在 $\alpha > 0$ 时有效，需要调参。开始建议不使用，即用默认值0，如果对L2的正则化不满意再加上L1正则化。

从上面可见，使用NMF的关键参数在于主题数的选择n_components和正则化的两个超参数 α, ρ 。

此外， W 矩阵一般在调用fit_transform方法的返回值里获得，而 H 矩阵则保存在NMF类的components_成员中。

下面我们给一个例子，我们有4个词，5个文本组成的矩阵，需要找出这些文本和隐含的两个主题之间的关系。代码如下：

```
import numpy as np
X = np.array([[1,1,5,2,3], [0,6,2,1,1], [3, 4, 0, 3, 1], [4, 1, 5, 6, 3]])
from sklearn.decomposition import NMF
model = NMF(n_components=2, alpha=0.01)
```

现在我们看看分解得到的 W, H ：

```
W = model.fit_transform(X)
H = model.components_
print W
print H
```

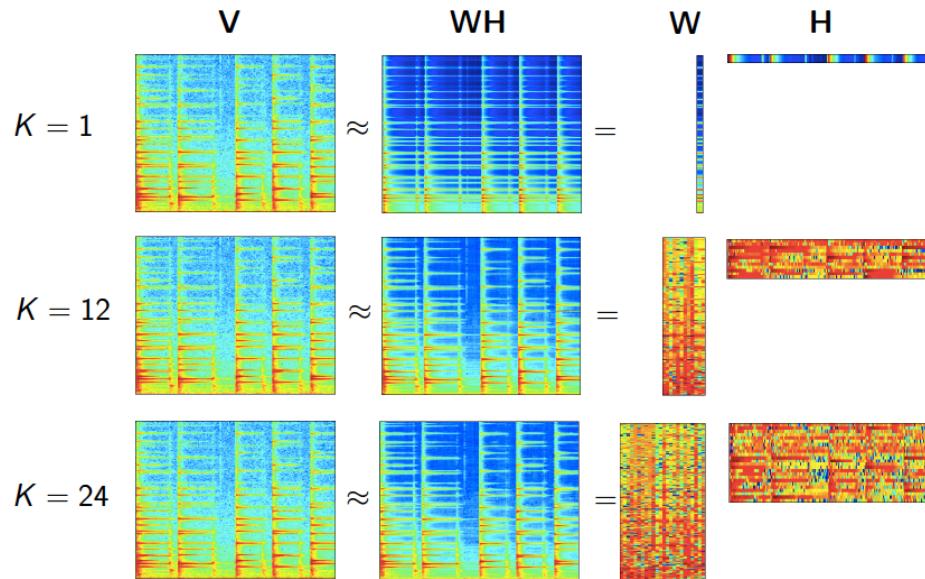
结果如下：

```
[[ 1.67371185  0.02013017]
 [ 0.40564826  2.17004352]
 [ 0.77627836  1.5179425 ]
 [ 2.66991709  0.00940262]]
[[ 1.32014421  0.40901559  2.10322743  1.99087019  1.29852389]
 [ 0.25859086  2.59911791  0.00488947  0.37089193  0.14622829]]
```

从结果可以看出，第1,3,4,5个文本和第一个隐含主题更相关，而第二个文本与第二个隐含主题更加相关。如果需要下一个结论，我们可以说，第1,3,4,5个文本属于第一个隐含主题，而第二个问题属于第2个隐含主题。

5. NMF的其他应用

虽然我们是在主题模型里介绍的NMF，但实际上NMF的适用领域很广，除了我们上面说的图像处理，语音处理，还包括信号处理与医药工程等，是一个普适的方法。在这些领域使用NMF的关键在于将NMF套入一个合适的模型，使得 W, H 矩阵都可以有明确的意义。这里给一个图展示NMF在做语音处理时的情形：



6. NMF主题模型小结

NMF作为一个漂亮的矩阵分解方法，它可以很好的用于主题模型，并且使主题的结果有基于概率分布的解释性。但是NMF以及它的变种pLSA虽然可以从概率的角度解释了主题模型，却都只能对训练样本中的文本进行主题识别，而对不在样本中的文本是无法识别其主题的。根本原因在于NMF与pLSA这类主题模型方法没有考虑主题概率分布的先验知识，比如文本中出现体育主题的概率肯定比哲学主题的概率要高，这点来源于我们的先验知识，但是无法告诉NMF主题模型。而LDA主题模型则考虑到了这一问题，目前来说，绝大多数的文本主题模型都是使用LDA以及其变体。下一篇我们就来讨论LDA主题模型。

文本主题模型之LDA(一) LDA基础

[文本主题模型之LDA\(一\) LDA基础](#)

[文本主题模型之LDA\(二\) LDA求解之Gibbs采样算法](#)

[文本主题模型之LDA\(三\) LDA求解之变分推断EM算法](#)

在前面我们讲到了基于矩阵分解的LSI和NMF主题模型，这里我们开始讨论被广泛使用的主题模型：隐含狄利克雷分布(Latent Dirichlet Allocation，以下简称LDA)。注意机器学习还有一个LDA，即线性判别分析，主要是用于降维和分类的，如果大家需要了解这个LDA的信息，参看之前写的[线性判别分析LDA原理总结](#)。文本关注于隐含狄利克雷分布对应的LDA。

1. LDA贝叶斯模型

LDA是基于贝叶斯模型的，涉及到贝叶斯模型离不开“先验分布”，“数据（似然）”和“后验分布”三块。在[朴素贝叶斯算法原理小结](#)中我们也已经讲到了这套贝叶斯理论。在贝叶斯学派这里：

$$\text{先验分布} + \text{数据（似然）} = \text{后验分布}$$

这点其实很好理解，因为这符合我们人的思维方式，比如你对好人和坏人的认知，先验分布为：100个好人和100个坏人，即你认为好人坏人各占一半，现在你被2个好人（数据）帮助了和1个坏人骗了，于是你得到了新的后验分布为：102个好人和101个的坏人。现在你的后验分布里面认为好人比坏人多了。这个后验分布接着又变成你的新的先验分布，当你被1个好人（数据）帮助了和3个坏人（数据）骗了后，你又更新了你的后验分布为：103个好人和104个的坏人。依次继续更新下去。

2. 二项分布与Beta分布

对于上一节的贝叶斯模型和认知过程，假如用数学和概率的方式该如何表达呢？

对于我们的数据（似然），这个好办，用一个二项分布就可以搞定，即对于二项分布：

$$Binom(k|n,p) = \binom{n}{k} p^k (1-p)^{n-k}$$

其中p我们可以理解为好人的概率，k为好人的个数，n为好人坏人的总数。

虽然数据（似然）很好理解，但是对于先验分布，我们就要费一番脑筋了，为什么呢？因为我们希望这个先验分布和数据（似然）对应的二项分布集合后，得到的后验分布在后面还可以作为先验分布！就像上面例子里的“102个好人和101个的坏人”，它是前面一次贝叶斯推荐的后验分布，又是后一次贝叶斯推荐的先验分布。也即是说，我们希望先验分布和后验分布的形式应该是一样的，这样的分布我们一般叫共轭分布。在我们的例子里，我们希望找到和二项分布共轭的分布。

和二项分布共轭的分布其实就是Beta分布。Beta分布的表达式为：

$$Beta(p|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1} (1-p)^{\beta-1}$$

其中 Γ 是Gamma函数，满足 $\Gamma(x) = (x - 1)!$

仔细观察Beta分布和二项分布，可以发现两者的密度函数很相似，区别仅仅在前面的归一化的阶乘项。那么它如何做到先验分布和后验分布的形式一样呢？后验分布 $P(p|n, k, \alpha, \beta)$ 推导如下：

$$\begin{aligned} P(p|n, k, \alpha, \beta) &\propto P(k|n, p)P(p|\alpha, \beta) & (1) \\ &= P(k|n, p)P(p|\alpha, \beta) & (2) \\ &= Binom(k|n, p)Beta(p|\alpha, \beta) & (3) \\ &= \binom{n}{k} p^k (1-p)^{n-k} \times \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} p^{\alpha-1} (1-p)^{\beta-1} & (4) \\ &\propto p^{k+\alpha-1} (1-p)^{n-k+\beta-1} & (5) \end{aligned}$$

将上面最后的式子归一化以后，得到我们的后验概率为：

$$P(p|n, k, \alpha, \beta) = \frac{\Gamma(\alpha + \beta + n)}{\Gamma(\alpha + k)\Gamma(\beta + n - k)} p^{k+\alpha-1} (1-p)^{n-k+\beta-1}$$

可见我们的后验分布的确是Beta分布，而且我们发现：

$$Beta(p|\alpha, \beta) + BinomCount(k, n - k) = Beta(p|\alpha + k, \beta + n - k)$$

这个式子完全符合我们在上一节好人坏人例子里的情况，我们的认知会把数据里的好人坏人分别加到我们的先验分布上，得到后验分布。

我们在来看看Beta分布 $Beta(p|\alpha, \beta)$ 的期望：

$$E(Beta(p|\alpha, \beta)) = \int_0^1 tBeta(p|\alpha, \beta) dt \quad (6)$$

$$= \int_0^1 t \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} t^{\alpha-1} (1-t)^{\beta-1} dt \quad (7)$$

$$= \int_0^1 \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} t^\alpha (1-t)^{\beta-1} dt \quad (8)$$

由于上式最右边的乘积对应Beta分布 $Beta(p|\alpha + 1, \beta)$ ，因此有：

$$\int_0^1 \frac{\Gamma(\alpha + \beta + 1)}{\Gamma(\alpha + 1)\Gamma(\beta)} p^\alpha (1-p)^{\beta-1} = 1$$

这样我们的期望可以表达为：

$$E(Beta(p|\alpha, \beta)) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{\Gamma(\alpha + 1)\Gamma(\beta)}{\Gamma(\alpha + \beta + 1)} = \frac{\alpha}{\alpha + \beta}$$

这个结果也很符合我们的思维方式。

3. 多项分布与Dirichlet 分布

现在我们回到上面好人坏人的问题，假如我们发现有第三类人，不好不坏的人，这时候我们如何用贝叶斯来表达这个模型分布呢？之前我们是二维分布，现在是三维分布。由于二维我们使用了Beta分布和二项分布来表达这个模型，则在三维时，以此类推，我们可以用三维的Beta分布来表达先验后验分布，三项的多项分布来表达数据（似然）。

三项的多项分布好表达，我们假设数据中的第一类有 m_1 个好人，第二类有 m_2 个坏人，第三类为 $m_3 = n - m_1 - m_2$ 个不好不坏的人，对应的概率分别为 $p_1, p_2, p_3 = 1 - p_1 - p_2$ ，则对应的多项分布为：

$$multi(m_1, m_2, m_3 | n, p_1, p_2, p_3) = \frac{n!}{m_1! m_2! m_3!} p_1^{m_1} p_2^{m_2} p_3^{m_3}$$

那三维的Beta分布呢？超过二维的Beta分布我们一般称之为狄利克雷(以下称为Dirichlet)分布。也可以说Beta分布是Dirichlet 分布在二维时的特殊形式。从二维的Beta分布表达式，我们很容易写出三维的Dirichlet分布如下：

$$Dirichlet(p_1, p_2, p_3 | \alpha_1, \alpha_2, \alpha_3) = \frac{\Gamma(\alpha_1 + \alpha_2 + \alpha_3)}{\Gamma(\alpha_1)\Gamma(\alpha_2)\Gamma(\alpha_3)} p_1^{\alpha_1-1} (p_2)^{\alpha_2-1} (p_3)^{\alpha_3-1}$$

同样的方法，我们可以写出4维，5维，。。。以及更高维的Dirichlet 分布的概率密度函数。为了简化表达式，我们用向量来表示概率和计数，这样多项分布可以表示为： $Dirichlet(\vec{p}|\vec{\alpha})$ ，而多项分布可以表示为： $multi(\vec{m}|n, \vec{p})$ 。

一般意义上的K维Dirichlet 分布表达式为：

$$Dirichlet(\vec{p}|\vec{\alpha}) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K p_k^{\alpha_k-1}$$

而多项分布和Dirichlet 分布也满足共轭关系，这样我们可以得到和上一节类似的结论：

$$Dirichlet(\vec{p}|\vec{\alpha}) + MultiCount(\vec{m}) = Dirichlet(\vec{p}|\vec{\alpha} + \vec{m})$$

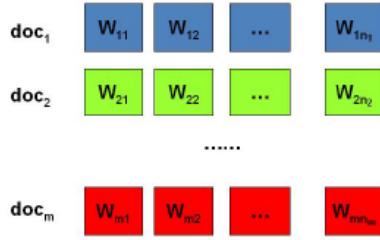
对于Dirichlet 分布的期望，也有和Beta分布类似的性质：

$$E(Dirichlet(\vec{p}|\vec{\alpha})) = \left(\frac{\alpha_1}{\sum_{k=1}^K \alpha_k}, \frac{\alpha_2}{\sum_{k=1}^K \alpha_k}, \dots, \frac{\alpha_K}{\sum_{k=1}^K \alpha_k} \right)$$

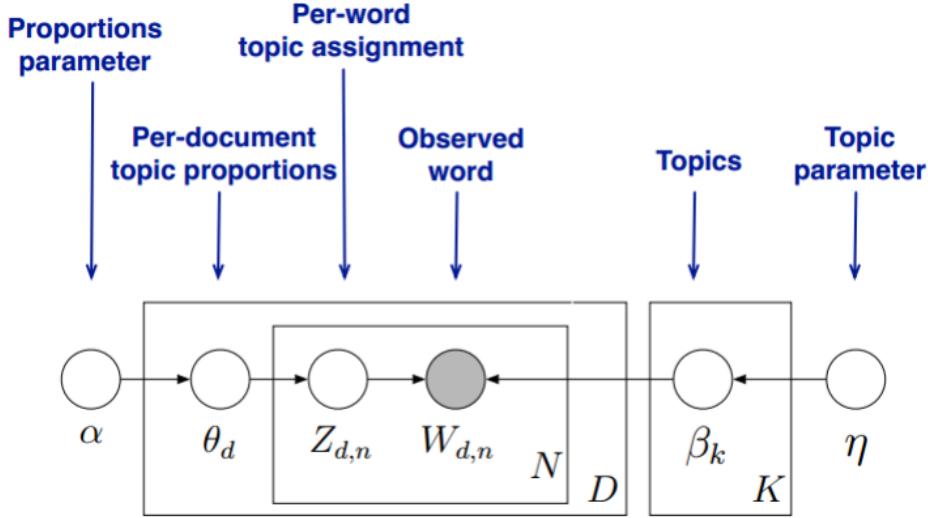
4. LDA主题模型

前面做了这么多的铺垫，我们终于可以开始LDA主题模型了。

我们的问题是这样的，我们有 M 篇文档，对应第 d 个文档中有 N_d 个词。即输入为如下图：



我们的目标是找到每一篇文档的主题分布和每一个主题中词的分布。在LDA模型中，我们需要先假定一个主题数目 K ，这样所有的分布就都基于 K 个主题展开。那么具体LDA模型是怎么样的呢？具体如下图：



LDA假设文档主题的先验分布是Dirichlet分布，即对于任一文档 d ，其主题分布 θ_d 为：

$$\theta_d = \text{Dirichlet}(\vec{\alpha})$$

其中， α 为分布的超参数，是一个 K 维向量。

LDA假设主题中词的先验分布是Dirichlet分布，即对于任一主题 k ，其词分布 β_k 为：

$$\beta_k = \text{Dirichlet}(\vec{\eta})$$

其中， η 为分布的超参数，是一个 V 维向量。 V 代表词汇表里所有词的个数。

对于数据中任一篇文档 d 中的第 n 个词，我们可以从主题分布 θ_d 中得到它的主题编号 z_{dn} 的分布为：

$$z_{dn} = \text{multi}(\theta_d)$$

而对于该主题编号，得到我们看到的词 w_{dn} 的概率分布为：

$$w_{dn} = \text{multi}(\beta_{z_{dn}})$$

理解LDA主题模型的主要任务就是理解上面的这个模型。这个模型里，我们有 M 个文档主题的Dirichlet分布，而对应的数据有 M 个主题编号的多项分布，这样 $(\alpha \rightarrow \theta_d \rightarrow z_d)$ 就组成了Dirichlet-multi共轭，可以使用前面提到的贝叶斯推断的方法得到基于Dirichlet分布的文档主题后验分布。

如果在第 d 个文档中，第 k 个主题的词的个数为： $n_d^{(k)}$ ，则对应的多项分布的计数可以表示为

$$\vec{n}_d = (n_d^{(1)}, n_d^{(2)}, \dots, n_d^{(K)})$$

利用Dirichlet-multi共轭，得到 θ_d 的后验分布为：

$$\text{Dirichlet}(\theta_d | \vec{\alpha} + \vec{n}_d)$$

同样的道理，对于主题与词的分布，我们有 K 个主题与词的Dirichlet分布，而对应的数据有 K 个主题编号的多项分布，这样 $(\eta \rightarrow \beta_k \rightarrow w_{(k)})$ 就组成了Dirichlet-multi共轭，可以使用前面提到的贝叶斯推断的方法得到基于Dirichlet分布的主题词的后验分布。

如果在第 k 个主题中，第 v 个词的个数为： $n_k^{(v)}$ ，则对应的多项分布的计数可以表示为

$$\vec{n}_k = (n_k^{(1)}, n_k^{(2)}, \dots, n_k^{(V)})$$

利用Dirichlet-multi共轭，得到 β_k 的后验分布为：

$$\text{Dirichlet}(\beta_k | \vec{\eta} + \vec{n}_k)$$

文本主题模型之LDA(二) LDA求解之Gibbs采样算法

[文本主题模型之LDA\(一\) LDA基础](#)

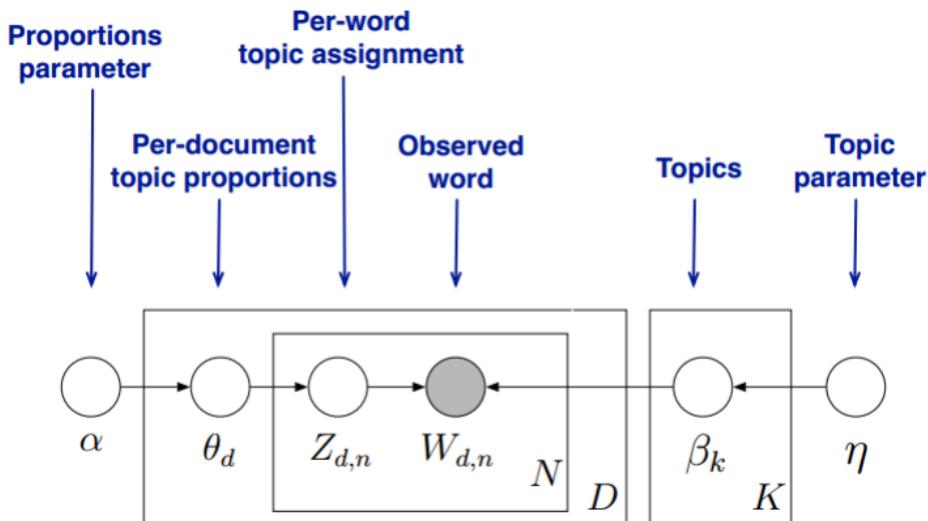
[文本主题模型之LDA\(二\) LDA求解之Gibbs采样算法](#)

[文本主题模型之LDA\(三\) LDA求解之变分推断EM算法](#)

本文是LDA主题模型的第二篇，读这一篇之前建议先读[文本主题模型之LDA\(一\) LDA基础](#)，同时由于使用了基于MCMC的Gibbs采样算法，如果你对MCMC和Gibbs采样不熟悉，建议阅读之前写的MCMC系列[MCMC\(四\) Gibbs采样](#)。

1. Gibbs采样算法求解LDA的思路

首先，回顾LDA的模型图如下：



在Gibbs采样算法求解LDA的方法中，我们的 α, η 是已知的先验输入，我们的目标是得到各个 z_{dn}, w_{kn} 对应的整体 \vec{z}, \vec{w} 的概率分布，即文档主题的分布和主题词的分布。由于我们是采用Gibbs采样法，则对于要求的目标分布，我们需要得到对应分布各个特征维度的条件概率分布。

具体到我们的问题，我们的所有文档联合起来形成的词向量 \vec{w} 是已知的数据，不知道的是语料库主题 \vec{z} 的分布。假如我们可以先求出 w, z 的联合分布 $p(\vec{w}, \vec{z})$ ，进而可以求出某一个词 w_i 对应主题特征 z_i 的条件概率分布 $p(z_i = k | \vec{w}, \vec{z}_{-i})$ 。其中， \vec{z}_{-i} 代表去掉下标为 i 的词后的主题分布。有了条件概率分布 $p(z_i = k | \vec{w}, \vec{z}_{-i})$ ，我们就可以进行Gibbs采样，最终在Gibbs采样收敛后得到第 i 个词的主题。

如果我们通过采样得到了所有词的主题，那么通过统计所有词的主题计数，就可以得到各个主题的词分布。接着统计各个文档对应词的主题计数，就可以得到各个文档的主题分布。

以上就是Gibbs采样算法求解LDA的思路。

2. 主题和词的联合分布与条件分布的求解

从上一节可以发现，要使用Gibbs采样求解LDA，关键是得到条件概率 $p(z_i = k | \vec{w}, \vec{z}_{-i})$ 的表达式。那么这一节我们的目标就是求出这个表达式供Gibbs采样使用。

首先我们简化下Dirichlet分布的表达式，其中 $\Delta(\alpha)$ 是归一化参数：

$$Dirichlet(\vec{p} | \vec{\alpha}) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K p_k^{\alpha_k - 1} = \frac{1}{\Delta(\vec{\alpha})} \prod_{k=1}^K p_k^{\alpha_k - 1}$$

现在我们先计算下第d个文档的主题的条件分布 $p(\vec{z}_d|\alpha)$ ，在上一篇中我们讲到 $\alpha \rightarrow \theta_d \rightarrow \vec{z}_d$ 组成了Dirichlet-multi共轭，利用这组分布，计算 $p(\vec{z}_d|\vec{\alpha})$ 如下：

$$p(\vec{z}_d|\vec{\alpha}) = \int p(\vec{z}_d|\vec{\theta}_d)p(\theta_d|\vec{\alpha})d\vec{\theta}_d \quad (1)$$

$$= \int \prod_{k=1}^K p_k^{n_d^{(k)}} \text{Dirichlet}(\vec{\alpha}) d\vec{\theta}_d \quad (2)$$

$$= \int \prod_{k=1}^K p_k^{n_d^{(k)}} \frac{1}{\Delta(\vec{\alpha})} \prod_{k=1}^K p_k^{\alpha_k-1} d\vec{\theta}_d \quad (3)$$

$$= \frac{1}{\Delta(\vec{\alpha})} \int \prod_{k=1}^K p_k^{n_d^{(k)} + \alpha_k - 1} d\vec{\theta}_d \quad (4)$$

$$= \frac{\Delta(\vec{n}_d + \vec{\alpha})}{\Delta(\vec{\alpha})} \quad (5)$$

其中，在第d个文档中，第k个主题的词的个数表示为： $n_d^{(k)}$ ，对应的多项分布的计数可以表示为

$$\vec{n}_d = (n_d^{(1)}, n_d^{(2)}, \dots, n_d^{(K)})$$

有了单一个文档的主题条件分布，则可以得到所有文档的主题条件分布为：

$$p(\vec{z}|\vec{\alpha}) = \prod_{d=1}^M p(\vec{z}_d|\vec{\alpha}) = \prod_{d=1}^M \frac{\Delta(\vec{n}_d + \vec{\alpha})}{\Delta(\vec{\alpha})}$$

同样的方法，可以得到，第k个主题对应的词的条件分布 $p(\vec{w}|\vec{z}, \vec{\eta})$ 为：

$$p(\vec{w}|\vec{z}, \vec{\eta}) = \prod_{k=1}^K p(\vec{w}_k|\vec{z}, \vec{\eta}) = \prod_{k=1}^K \frac{\Delta(\vec{n}_k + \vec{\eta})}{\Delta(\vec{\eta})}$$

其中，第k个主题中，第v个词的个数表示为： $n_k^{(v)}$ ，对应的多项分布的计数可以表示为

$$\vec{n}_k = (n_k^{(1)}, n_k^{(2)}, \dots, n_k^{(V)})$$

最终我们得到主题和词的联合分布 $p(\vec{w}, \vec{z}|\vec{\alpha}, \vec{\eta})$ 如下：

$$p(\vec{w}, \vec{z}) \propto p(\vec{w}, \vec{z}|\vec{\alpha}, \vec{\eta}) = p(\vec{z}|\vec{\alpha})p(\vec{w}|\vec{z}, \vec{\eta}) = \prod_{d=1}^M \frac{\Delta(\vec{n}_d + \vec{\alpha})}{\Delta(\vec{\alpha})} \prod_{k=1}^K \frac{\Delta(\vec{n}_k + \vec{\eta})}{\Delta(\vec{\eta})}$$

有了联合分布，现在我们就可以求Gibbs采样需要的条件分布 $p(z_i = k|\vec{w}, \vec{z}_{-i})$ 了。需要注意的是这里的i是一个二维下标，对应第d篇文档的第n个词。

对于下标*i*，由于它对应的词 w_i 是可以观察到的，因此我们有：

$$p(z_i = k|\vec{w}, \vec{z}_{-i}) \propto p(z_i = k, w_i = t|\vec{w}_{-i}, \vec{z}_{-i})$$

对于 $z_i = k, w_i = t$ ，它只涉及到第d篇文档和第k个主题两个Dirichlet-multi共轭，即：

$$\vec{\alpha} \rightarrow \vec{\theta}_d \rightarrow \vec{z}_d$$

$$\vec{\eta} \rightarrow \vec{\beta}_k \rightarrow \vec{w}_{(k)}$$

其余的 $M + K - 2$ 个Dirichlet-multi共轭和它们这两个共轭是独立的。如果我们在语料库中去掉 z_i, w_i ，并不会改变之前的 $M + K$ 个Dirichlet-multi共轭结构，只是向量的某些位置的计数会减少，因此对于 $\vec{\theta}_d, \vec{\beta}_k$ ，对应的后验分布为：

$$p(\vec{\theta}_d|\vec{w}_{-i}, \vec{z}_{-i}) = \text{Dirichlet}(\vec{\theta}_d|\vec{n}_{d,-i} + \vec{\alpha})$$

$$p(\vec{\beta}_k|\vec{w}_{-i}, \vec{z}_{-i}) = \text{Dirichlet}(\vec{\beta}_k|\vec{n}_{k,-i} + \vec{\eta})$$

现在开始计算Gibbs采样需要的条件概率：

$$p(z_i = k|\vec{w}, \vec{z}_{-i}) \propto p(z_i = k, w_i = t|\vec{w}_{-i}, \vec{z}_{-i}) \quad (6)$$

$$= \int p(z_i = k, w_i = t, \vec{\theta}_d, \vec{\beta}_k|\vec{w}_{-i}, \vec{z}_{-i}) d\vec{\theta}_d d\vec{\beta}_k \quad (7)$$

$$= \int p(z_i = k, \vec{\theta}_d|\vec{w}_{-i}, \vec{z}_{-i}) p(w_i = t, \vec{\beta}_k|\vec{w}_{-i}, \vec{z}_{-i}) d\vec{\theta}_d d\vec{\beta}_k \quad (8)$$

$$= \int p(z_i = k|\vec{\theta}_d) p(\vec{\theta}_d|\vec{w}_{-i}, \vec{z}_{-i}) p(w_i = t|\vec{\beta}_k) p(\vec{\beta}_k|\vec{w}_{-i}, \vec{z}_{-i}) d\vec{\theta}_d d\vec{\beta}_k \quad (9)$$

$$= \int p(z_i = k|\vec{\theta}_d) \text{Dirichlet}(\vec{\theta}_d|\vec{n}_{d,-i} + \vec{\alpha}) d\vec{\theta}_d \quad (10)$$

$$* \int p(w_i = t|\vec{\beta}_k) \text{Dirichlet}(\vec{\beta}_k|\vec{n}_{k,-i} + \vec{\eta}) d\vec{\beta}_k \quad (11)$$

$$= \int \theta_{dk} \text{Dirichlet}(\vec{\theta}_d|\vec{n}_{d,-i} + \vec{\alpha}) d\vec{\theta}_d \int \beta_{kt} \text{Dirichlet}(\vec{\beta}_k|\vec{n}_{k,-i} + \vec{\eta}) d\vec{\beta}_k \quad (12)$$

$$= E_{\text{Dirichlet}(\theta_{di})}(\theta_{dk}) E_{\text{Dirichlet}(\beta_{ki})}(\beta_{kt}) \quad (13)$$

在上一篇LDA基础里我们讲到了Dirichlet分布的期望公式，因此我们有：

$$E_{Dirichlet(\theta_d)}(\theta_{dk}) = \frac{n_{d,-i}^k + \alpha_k}{\sum_{s=1}^K n_{d,-i}^s + \alpha_s}$$

$$E_{Dirichlet(\beta_k)}(\beta_{kt}) = \frac{n_{k,-i}^t + \eta_t}{\sum_{f=1}^V n_{k,-i}^f + \eta_f}$$

最终我们得到每个词对应主题的Gibbs采样的条件概率公式为：

$$p(z_i = k | \vec{w}, \vec{z}_{-i}) = \frac{n_{d,-i}^k + \alpha_k}{\sum_{s=1}^K n_{d,-i}^s + \alpha_s} \frac{n_{k,-i}^t + \eta_t}{\sum_{f=1}^V n_{k,-i}^f + \eta_f}$$

有了这个公式，我们就可以用Gibbs采样去采样所有词的主题，当Gibbs采样收敛后，即得到所有词的采样主题。

利用所有采样得到的词和主题的对应关系，我们就可以得到每个文档词主题的分布 θ_d 和每个主题中所有词的分布 β_k 。

3. LDA Gibbs采样算法流程总结

现在我们总结下LDA Gibbs采样算法流程。首先是训练流程：

- 1) 选择合适的主题数 K , 选择合适的超参数向量 $\vec{\alpha}, \vec{\eta}$
- 2) 对应语料库中每一篇文档的每一个词，随机的赋予一个主题编号 z
- 3) 重新扫描语料库，对于每一个词，利用Gibbs采样公式更新它的topic编号，并更新语料库中该词的编号。
- 4) 重复第2步的基于坐标轴轮换的Gibbs采样，直到Gibbs采样收敛。
- 5) 统计语料库中的各个文档各个词的主题，得到文档主题分布 θ_d ，统计语料库中各个主题词的分布，得到LDA的主题与词的分布 β_k 。

下面我们再来看看当新文档出现时，如何统计该文档的主题。此时我们的模型已定，也就是LDA的各个主题的词分布 β_k 已经确定，我们需要得到的是该文档的主题分布。因此在Gibbs采样时，我们的 $E_{Dirichlet(\beta_k)}(\beta_{kt})$ 已经固定，只需要对前半部分 $E_{Dirichlet(\theta_d)}(\theta_{dk})$ 进行采样计算即可。

现在我们总结下LDA Gibbs采样算法的预测流程：

- 1) 对应当前文档的每一个词，随机的赋予一个主题编号 z
- 2) 重新扫描当前文档，对于每一个词，利用Gibbs采样公式更新它的topic编号。
- 3) 重复第2步的基于坐标轴轮换的Gibbs采样，直到Gibbs采样收敛。
- 4) 统计文档中各个词的主题，得到该文档主题分布。

4. LDA Gibbs采样算法小结

使用Gibbs采样算法训练LDA模型，我们需要先确定三个超参数 $K, \vec{\alpha}, \vec{\eta}$ 。其中选择一个合适的 K 尤其关键，这个值一般和我们解决问题的目的有关。如果只是简单的语义区分，则较小的 K 即可，如果是复杂的语义区分，则 K 需要较大，而且还需要足够的语料。

由于Gibbs采样可以很容易的并行化，因此也可以很方便的使用大数据平台来分布式的训练海量文档的LDA模型。以上就是LDA Gibbs采样算法。

后面我们会介绍用变分推断EM算法来求解LDA主题模型，这个方法是scikit-learn和spark MLlib都使用的LDA求解方法。

文本主题模型之LDA(三) LDA求解之变分推断EM算法

[文本主题模型之LDA\(一\) LDA基础](#)

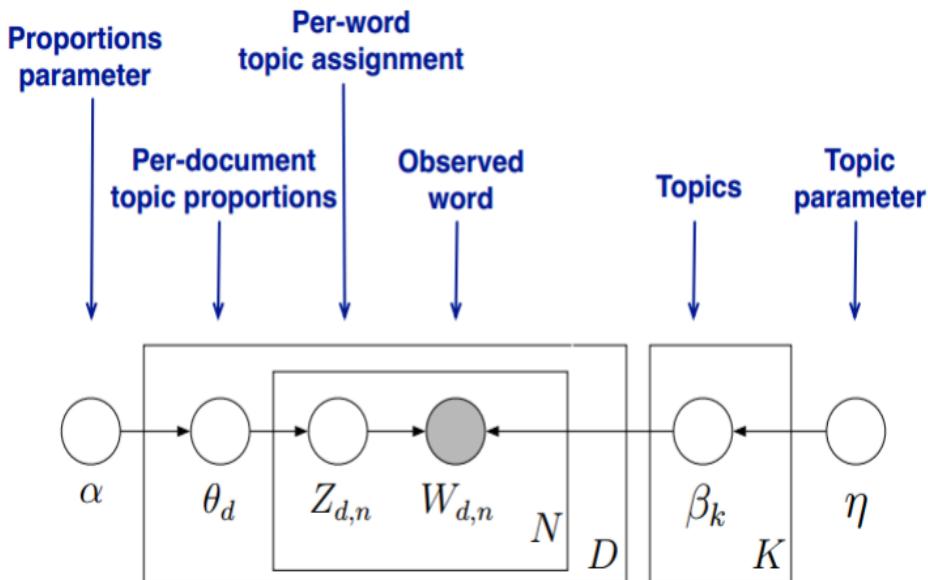
[文本主题模型之LDA\(二\) LDA求解之Gibbs采样算法](#)

[文本主题模型之LDA\(三\) LDA求解之变分推断EM算法](#)

本文是LDA主题模型的第三篇，读这一篇之前建议先读[文本主题模型之LDA\(一\) LDA基础](#)，同时由于使用了EM算法，如果你对EM算法不熟悉，建议先熟悉EM算法的主要思想。LDA的变分推断EM算法求解，应用于Spark MLlib和Scikit-learn的LDA算法实现，因此值得好好理解。

1. 变分推断EM算法求解LDA的思路

首先，回顾LDA的模型图如下：



变分推断EM算法希望通过“变分推断(Variational Inference)”和EM算法来得到LDA模型的文档主题分布和主题词分布。首先来看EM算法在这里的使用，我们的模型里面有隐藏变量 θ, β, z ，模型的参数是 α, η 。为了求出模型参数和对应的隐藏变量分布，EM算法需要在E步先求出隐藏变量 θ, β, z 的基于条件概率分布的期望，接着在M步极大化这个期望，得到更新的后验模型参数 α, η 。

问题是在EM算法的E步，由于 θ, β, z 的耦合，我们难以求出隐藏变量 θ, β, z 的条件概率分布，也难以求出对应的期望，需要“变分推断”来帮忙，这里所谓的变分推断，也就是在隐藏变量存在耦合的情况下，我们通过变分假设，即假设所有的隐藏变量都是通过各自的独立分布形成的，这样就去掉了隐藏变量之间的耦合关系。我们用各个独立分布形成的变分分布来模拟近似隐藏变量的条件分布，这样就可以顺利的使用EM算法了。

当进行若干轮的E步和M步的迭代更新之后，我们可以得到合适的近似隐藏变量分布 θ, β, z 和模型后验参数 α, η ，进而就得到了我们需要的LDA文档主题分布和主题词分布。

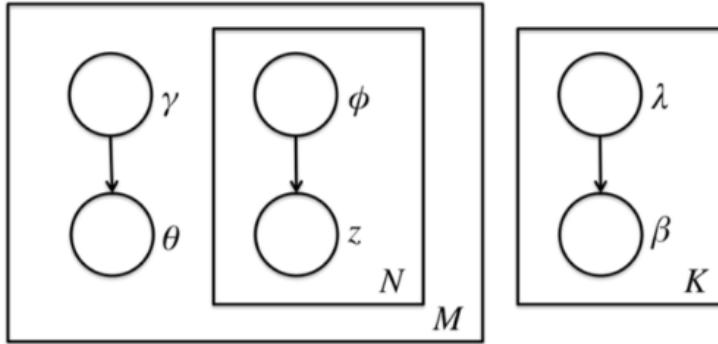
可见要完全理解LDA的变分推断EM算法，需要搞清楚它在E步变分推断的过程和推断完毕后EM算法的过程。

2. LDA的变分推断思路

要使用EM算法，我们需要求出隐藏变量的条件概率分布如下：

$$p(\theta, \beta, z | w, \alpha, \eta) = \frac{p(\theta, \beta, z, w | \alpha, \eta)}{p(w | \alpha, \eta)}$$

前面讲到由于 θ, β, z 之间的耦合，这个条件概率是没法直接求的，但是如果不要求它就不能用EM算法了。怎么办呢，我们引入变分推断，具体是引入基于mean field assumption的变分推断，这个推断假设所有的隐藏变量都是通过各自的独立分布形成的，如下图所示：



我们假设隐藏变量 θ 是由独立分布 γ 形成的，隐藏变量 z 是由独立分布 ϕ 形成的，隐藏变量 β 是由独立分布 λ 形成的。这样我们得到了三个隐藏变量联合的变分分布 q 为：

$$q(\beta, z, \theta | \lambda, \phi, \gamma) = \prod_{k=1}^K q(\beta_k | \lambda_k) \prod_{d=1}^M q(\theta_d, z_d | \gamma_d, \phi_d) \quad (1)$$

$$= \prod_{k=1}^K q(\beta_k | \lambda_k) \prod_{d=1}^M (q(\theta_d | \gamma_d) \prod_{n=1}^{N_d} q(z_{dn} | \phi_{dn})) \quad (2)$$

我们的目标是用 $q(\beta, z, \theta | \lambda, \phi, \gamma)$ 来近似的估计 $p(\theta, \beta, z | w, \alpha, \eta)$ ，也就是说需要这两个分布尽可能的相似，用数学语言来描述就是希望这两个概率分布之间有尽可能小的KL距离，即：

$$(\lambda^*, \phi^*, \gamma^*) = \underbrace{\arg \min_{\lambda, \phi, \gamma}}_{\lambda, \phi, \gamma} D(q(\beta, z, \theta | \lambda, \phi, \gamma) || p(\theta, \beta, z | w, \alpha, \eta))$$

其中 $D(q||p)$ 即为KL散度或KL距离，对应分布 q 和 p 的交叉熵。即：

$$D(q||p) = \sum_x q(x) \log \frac{q(x)}{p(x)} = E_{q(x)}(\log q(x) - \log p(x))$$

我们的目的就是找到合适的 $\lambda^*, \phi^*, \gamma^*$ ，然后用 $q(\beta, z, \theta | \lambda^*, \phi^*, \gamma^*)$ 来近似隐藏变量的条件分布 $p(\theta, \beta, z | w, \alpha, \eta)$ ，进而使用EM算法迭代。

这个合适的 $\lambda^*, \phi^*, \gamma^*$ 也不是那么好求的，怎么办呢？我们先看看我能文档数据的对数似然函数 $\log(w|\alpha, \eta)$ 如下，为了简化表示，我们用 $E_q(x)$ 代替 $E_{q(\beta, z, \theta | \lambda, \phi, \gamma)}(x)$ ，用来表示 x 对于变分分布 $q(\beta, z, \theta | \lambda, \phi, \gamma)$ 的期望。

$$\log(w|\alpha, \eta) = \log \int \int \sum_z p(\theta, \beta, z, w | \alpha, \eta) d\theta d\beta \quad (3)$$

$$= \log \int \int \sum_z \frac{p(\theta, \beta, z, w | \alpha, \eta) q(\beta, z, \theta | \lambda, \phi, \gamma)}{q(\beta, z, \theta | \lambda, \phi, \gamma)} d\theta d\beta \quad (4)$$

$$= \log E_q \frac{p(\theta, \beta, z, w | \alpha, \eta)}{q(\beta, z, \theta | \lambda, \phi, \gamma)} \quad (5)$$

$$\geq E_q \log \frac{p(\theta, \beta, z, w | \alpha, \eta)}{q(\beta, z, \theta | \lambda, \phi, \gamma)} \quad (6)$$

$$= E_q \log p(\theta, \beta, z, w | \alpha, \eta) - E_q \log q(\beta, z, \theta | \lambda, \phi, \gamma) \quad (7)$$

其中，从第(5)式到第(6)式用到了Jensen不等式：

$$f(E(x)) \geq E(f(x)) \quad f(x) \text{ 为凹函数}$$

我们一般把第(7)式记为：

$$L(\lambda, \phi, \gamma; \alpha, \eta) = E_q \log p(\theta, \beta, z, w | \alpha, \eta) - E_q \log q(\beta, z, \theta | \lambda, \phi, \gamma)$$

由于 $L(\lambda, \phi, \gamma; \alpha, \eta)$ 是我们的对数似然的一个下界（第6式），所以这个 L 一般称为ELBO(Evidence Lower BOund)。那么这个ELBO和我们需要优化的KL散度有什么关系呢？注意到：

$$D(q(\beta, z, \theta | \lambda, \phi, \gamma) || p(\theta, \beta, z | w, \alpha, \eta)) = E_q \log q(\beta, z, \theta | \lambda, \phi, \gamma) - E_q \log p(\theta, \beta, z | w, \alpha, \eta) \quad (8)$$

$$= E_q \log q(\beta, z, \theta | \lambda, \phi, \gamma) - E_q \log \frac{p(\theta, \beta, z, w | \alpha, \eta)}{p(w | \alpha, \eta)} \quad (9)$$

$$= -L(\lambda, \phi, \gamma; \alpha, \eta) + \log(w | \alpha, \eta) \quad (10)$$

在(10)式中，由于对数似然部分和我们的KL散度无关，可以看做常量，因此我们希望最小化KL散度等价于最大化ELBO。那么我们的变分推断最终等价的转化为要求ELBO的最大值。现在我们开始关注于极大化ELBO并求出极值对应的变分参数 λ, ϕ, γ 。

3. 极大化ELBO求解变分参数

为了极大化ELBO，我们首先对ELBO函数做一个整理如下：

$$L(\lambda, \phi, \gamma; \alpha, \eta) = E_q[\log p(\beta|\eta)] + E_q[\log p(z|\theta)] + E_q[\log p(\theta|\alpha)] \quad (11)$$

$$+ E_q[\log p(w|z, \beta)] - E_q[\log q(\beta|\lambda)] \quad (12)$$

$$- E_q[\log q(z|\phi)] - E_q[\log q(\theta|\gamma)] \quad (13)$$

可见展开后有7项，现在我们需要对这7项分别做一个展开。为了简化篇幅，这里只对第一项的展开做详细介绍。在介绍第一项的展开前，我们需要了解指数分布族的性质。指数分布族是指下面这样的概率分布：

$$p(x|\theta) = h(x) \exp(\eta(\theta) * T(x) - A(\theta))$$

其中， $A(x)$ 为归一化因子，主要是保证概率分布累积求和后为1，引入指数分布族主要是它有下面这样的性质：

$$\frac{d}{d\eta(\theta)} A(\theta) = E_{p(x|\theta)}[T(x)]$$

这个证明并不复杂，这里不累述。我们的常见分布比如Gamma分布，Beta分布，Dirichlet分布都是指数分布族。有了这个性质，意味着我们在ELBO里面一大推的期望表达式可以转化为求导来完成，这个技巧大大简化了计算量。

回到我们ELBO第一项的展开如下：

$$E_q[\log p(\beta|\eta)] = E_q[\log \prod_{k=1}^K \left(\frac{\Gamma(\sum_{i=1}^V \eta_i)}{\prod_{i=1}^V \Gamma(\eta_i)} \prod_{i=1}^V \beta_{ki}^{\eta_i-1} \right)] \quad (14)$$

$$= K \log \Gamma(\sum_{i=1}^V \eta_i) - K \sum_{i=1}^V \log \Gamma(\eta_i) + \sum_{k=1}^K E_q[\sum_{i=1}^V (\eta_i - 1) \log \beta_{ki}] \quad (15)$$

第(15)式的第三项的期望部分，可以用上面讲到的指数分布族的性质，转化为一个求导过程。即：

$$E_q[\sum_{i=1}^V \log \beta_{ki}] = (\log \Gamma(\lambda_{ki}) - \log \Gamma(\sum_{i'=1}^V \lambda_{ki'}))' = \Psi(\lambda_{ki}) - \Psi(\sum_{i'=1}^V \lambda_{ki'})$$

其中：

$$\Psi(x) = \frac{d}{dx} \log \Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

最终，我们得到EBLO第一项的展开式为：

$$E_q[\log p(\beta|\eta)] = K \log \Gamma(\sum_{i=1}^V \eta_i) - K \sum_{i=1}^V \log \Gamma(\eta_i) + \sum_{k=1}^K \sum_{i=1}^V (\eta_i - 1) (\Psi(\lambda_{ki}) - \Psi(\sum_{i'=1}^V \lambda_{ki'})) \quad (16)$$

类似的方法求解其他6项，可以得到ELBO的最终关于变分参数 λ, ϕ, γ 的表达式。其他6项的表达式为：

$$E_q[\log p(z|\theta)] = \sum_{n=1}^N \sum_{k=1}^K \phi_{nk} \Psi(\gamma_k) - \Psi(\sum_{k'=1}^K \gamma_{k'}) \quad (17)$$

$$E_q[\log p(\theta|\alpha)] = \log \Gamma(\sum_{k=1}^K \alpha_k) - \sum_{k=1}^K \log \Gamma(\alpha_k) + \sum_{k=1}^K (\alpha_k - 1) (\Psi(\gamma_k) - \Psi(\sum_{k'=1}^K \gamma_{k'})) \quad (18)$$

$$E_q[\log p(w|z, \beta)] = \sum_{n=1}^N \sum_{k=1}^K \sum_{i=1}^V \phi_{nk} w_n^i (\Psi(\lambda_{ki}) - \Psi(\sum_{i'=1}^V \lambda_{ki'})) \quad (19)$$

$$E_q[\log q(\beta|\lambda)] = \sum_{k=1}^K (\log \Gamma(\sum_{i=1}^V \lambda_{ki}) - \sum_{i=1}^V \log \Gamma(\lambda_{ki})) + \sum_{k=1}^K \sum_{i=1}^V (\lambda_{ki} - 1) (\Psi(\lambda_{ki}) - \Psi(\sum_{i'=1}^V \lambda_{ki'})) \quad (20)$$

$$E_q[\log q(z|\phi)] = \sum_{n=1}^N \sum_{k=1}^K \phi_{nk} \log \phi_{nk} \quad (21)$$

$$E_q[\log q(\theta|\gamma)] = \log \Gamma(\sum_{k=1}^K \gamma_k) - \sum_{k=1}^K \log \Gamma(\gamma_k) + \sum_{k=1}^K (\gamma_k - 1) (\Psi(\gamma_k) - \Psi(\sum_{k'=1}^K \gamma_{k'})) \quad (22)$$

有了ELBO的具体的关于变分参数 λ, ϕ, γ 的表达式，我们就可以用EM算法来迭代更新变分参数和模型参数了。

4. EM算法之E步：获取最优变分参数

有了前面变分推断得到的ELBO函数为基础，我们就可以进行EM算法了。但是和EM算法不同的是这里的E步需要在包含期望的EBLO计算最佳的变分参数。如何求解最佳的变分参数呢？通过对ELBO函数对各个变分参数 λ, ϕ, γ 分别求导并令偏导数为0，可以得到迭代表达式，多次迭代收敛后即为最佳变分参数。

这里就不详细推导了，直接给出各个变分参数的表达式如下：

$$\phi_{nk} \propto \exp(\sum_{i=1}^V w_n^i (\Psi(\lambda_{ki}) - \Psi(\sum_{i'=1}^V \lambda_{ki'})) + \Psi(\gamma_k) - \Psi(\sum_{k'=1}^K \gamma_{k'})) \quad (23)$$

其中， $w_n^i = 1$ 当且仅当文档中第*n*个词为词汇表中第*i*个词。

$$\gamma_k = \alpha_k + \sum_{n=1}^N \phi_{nk} \quad (24)$$

$$\lambda_{ki} = \eta_i + \sum_{n=1}^N \phi_{nk} w_n^i \quad (25)$$

由于变分参数 λ 决定了 β 的分布，对于整个语料是共有的，因此我们有：

$$\lambda_{ki} = \eta_i + \sum_{d=1}^M \sum_{n=1}^{N_d} \phi_{dnk} w_n^i \quad (26)$$

最终我们的E步就是用(23) (24) (26)式来更新三个变分参数。当我们得到三个变分参数后，不断循环迭代更新，直到这三个变分参数收敛。当变分参数收敛后，下一步就是M步，固定变分参数，更新模型参数 α, η 了。

5. EM算法之M步：更新模型参数

由于我们在E步，已经得到了当前最佳变分参数，现在我们在M步就来固定变分参数，极大化ELBO得到最优的模型参数 α, η 。求解最优的模型参数 α, η 的方法有很多，梯度下降法，牛顿法都可以。LDA这里一般使用的是牛顿法，即通过求出ELBO对于 α, η 的一阶导数和二阶导数的表达式，然后迭代求解 α, η 在M步的最优解。

对于 α ，它的一阶导数和二阶导数的表达式为：

$$\begin{aligned} \nabla_{\alpha_k} L &= M(\Psi(\sum_{k'=1}^K \alpha_{k'}) - \Psi(\alpha_k)) + \sum_{d=1}^M (\Psi'(\gamma_{dk}) - \Psi'(\sum_{k'=1}^K \gamma_{dk'})) \\ \nabla_{\alpha_k \alpha_j} L &= M(\Psi'(\sum_{k'=1}^K \alpha_{k'}) - \delta(k, j)\Psi'(\alpha_k)) \end{aligned}$$

其中，当且仅当 $k = j$ 时， $\delta(k, j) = 1$ ，否则 $\delta(k, j) = 0$ 。

对于 η ，它的一阶导数和二阶导数的表达式为：

$$\begin{aligned} \nabla_{\eta_i} L &= K(\Psi(\sum_{i'=1}^V \eta_{i'}) - \Psi(\eta_i)) + \sum_{k=1}^K (\Psi'(\lambda_{ki}) - \Psi'(\sum_{i'=1}^V \lambda_{ki'})) \\ \nabla_{\eta_i \eta_j} L &= K(\Psi'(\sum_{i'=1}^V \eta_{i'}) - \delta(i, j)\Psi'(\eta_i)) \end{aligned}$$

其中，当且仅当 $i = j$ 时， $\delta(i, j) = 1$ ，否则 $\delta(i, j) = 0$ 。

最终牛顿法迭代公式为：

$$\alpha_{k+1} = \alpha_k + \frac{\nabla_{\alpha_k} L}{\nabla_{\alpha_k \alpha_j} L} \quad (27)$$

$$\eta_{i+1} = \eta_i + \frac{\nabla_{\eta_i} L}{\nabla_{\eta_i \eta_j} L} \quad (28)$$

6. LDA变分推断EM算法流程总结

下面总结下LDA变分推断EM的算法的概要流程。

输入：主题数 K , M 个文档与对应的词。

1) 初始化 α, η 向量。

2) 开始EM算法迭代循环直到收敛。

a) 初始化所有的 ϕ, γ, λ ，进行LDA的E步迭代循环，直到 λ, ϕ, γ 收敛。

(i) for d from 1 to M:

for n from 1 to N_d :

for k from 1 to K:

按照(23)式更新 ϕ_{nk}

标准化 ϕ_{nk} 使该向量各项的和为1。

按照(24)式更新 γ_{ik} 。

(ii) for k from 1 to K:

for i from 1 to V:

按照(26)式更新 λ_{ki} 。

(iii)如果 ϕ, γ, λ 均已收敛，则跳出a)步，否则回到(i)步。

b) 进行LDA的M步迭代循环，直到 α, η 收敛

(i) 按照(27)(28)式用牛顿法迭代更新 α, η 直到收敛

c) 如果所有的参数均收敛，则算法结束，否则回到第2)步。

算法结束后，我们可以得到模型的后验参数 α, η ，以及我们需要的近似模型主题词分布 λ ，以及近似训练文档主题分布 γ 。

用scikit-learn学习LDA主题模型

在LDA模型原理篇我们总结了LDA主题模型的原理，这里我们就从应用的角度来使用scikit-learn来学习LDA主题模型。除了scikit-learn，还有spark MLlib和gensim库也有LDA主题模型的类库，使用的原理基本类似，本文关注于scikit-learn中LDA主题模型的使用。

1. scikit-learn LDA主题模型概述

在scikit-learn中，LDA主题模型的类在sklearn.decomposition.LatentDirichletAllocation包中，其算法实现主要基于原理篇里讲的变分推断EM算法，而没有使用基于Gibbs采样的MCMC算法实现。

而具体到变分推断EM算法，scikit-learn除了我们原理篇里讲到的标准的变分推断EM算法外，还实现了另一种在线变分推断EM算法，它在原理篇里的变分推断EM算法的基础上，为了避免文档内容太多太大而超过内存大小，而提供了分步训练(partial_fit函数)，即一次训练一小批样本文档，逐步更新模型，最终得到所有文档LDA模型的方法。这个改进算法我们没有讲，具体论文在这：“[Online Learning for Latent Dirichlet Allocation](#)”。

下面我们来看看sklearn.decomposition.LatentDirichletAllocation类库的主要参数。

2. scikit-learn LDA主题模型主要参数和方法

我们来看看LatentDirichletAllocation类的主要输入参数：

1) **n_topics**: 即我们的隐含主题数 K ，需要调参。 K 的大小取决于我们对主题划分的需求，比如我们只需要类似区分是动物，植物，还是非生物这样的粗粒度需求，那么 K 值可以取的很小，个位数即可。如果我们的目标是类似区分不同的动物以及不同的植物，不同的非生物这样的细粒度需求，则 K 值需要取的很大，比如上千上万。此时要求我们的训练文档数量要非常的多。

2) **doc_topic_prior**: 即我们的文档主题先验Dirichlet分布 θ_d 的参数 α 。一般如果我们没有主题分布的先验知识，可以使用默认值 $1/K$ 。

3) **topic_word_prior**: 即我们的主题词先验Dirichlet分布 β_k 的参数 η 。一般如果我们没有主题分布的先验知识，可以使用默认值 $1/K$ 。

4) **learning_method**: 即LDA的求解算法。有‘batch’ 和 ‘online’两种选择。‘batch’即我们在原理篇讲的变分推断EM算法，而“online”即在线变分推断EM算法，在“batch”的基础上引入了分步训练，将训练样本分批，逐步一批批的用样本更新主题词分布的算法。默认是“online”。选择了‘online’则我们可以在训练时使用partial_fit函数分布训练。不过在scikit-learn 0.20版本中默认算法会改回到“batch”。建议样本量不大只是用来学习的话用“batch”比较好，这样可以少很多参数要调。而样本太多太大的话，“online”则是首先了。

5) **learning_decay** : 仅仅在算法使用“online”时有意义，取值最好在(0.5, 1.0]，以保证“online”算法渐进的收敛。主要控制“online”算法的学习率，默认是0.7。一般不用修改这个参数。

6) **learning_offset** : 仅仅在算法使用“online”时有意义，取值要大于1。用来减小前面训练样本批次对最终模型的影响。

7) **max_iter** : EM算法的最大迭代次数。

8) **total_samples** : 仅仅在算法使用“online”时有意义，即分步训练时每一批文档样本的数量。在使用partial_fit函数时需要。

9) **batch_size**: 仅仅在算法使用“online”时有意义，即每次EM算法迭代时使用的文档样本的数量。

10) **mean_change_tol** : 即E步更新变分参数的阈值，所有变分参数更新小于阈值则E步结束，转入M步。一般不用修改默认值。

11) **max_doc_update_iter**: 即E步更新变分参数的最大迭代次数，如果E步迭代次数达到阈值，则转入M步。

从上面可以看出，如果learning_method使用“batch”算法，则需要注意的参数较少，则如果使用“online”，则需要注意“learning_decay”，“learning_offset”，“total_samples”和“batch_size”等参数。无论是“batch”还是“online”，n_topics(K)，doc_topic_prior(α)，topic_word_prior(η)都要注意。如果没有先验知识，则主要关注与主题数 K 。可以说，主题数 K 是LDA主题模型最重要的超参数。

3. scikit-learn LDA中文主题模型实例

下面我们给一个LDA中文主题模型的简单实例，从分词一直到LDA主题模型。

我们的有下面三段文档语料，分别放在了nlp_test0.txt, nlp_test2.txt和nlp_test4.txt：

沙瑞金赞叹易学习的胸怀，是金山的百姓有福，可是这件事对李达康的触动很大。易学习又回忆起他们三人分开的前一晚，大家一起喝酒话别，易学习被降职到道口县当县长，王大路下海经商，李达康连连赔礼道歉，觉得对不起大家，他最对不起的是王大路，就和易学习一起给王大路凑了5万块钱，王大路自己东挪西撮了5万块，开始下海经商。没想到后来王大路竟然做得风生水起。沙瑞金觉得他们三人，在困难时期还能以沫相助，很不容易。

沙瑞金向毛娅打听他们家在京州的别墅，毛娅笑着说，王大路事业有成之后，要给欧阳菁和她公司的股权，她们没有要，王大路就在京州帝豪园买了三套别墅，可是李达康和易学习都不要，这些房子都在王大路的名下，欧阳菁好像去住过，毛娅不想去，她觉得房子太大很浪费，自己家住得就很踏实。

347年（永和三年）三月，桓温兵至彭模（今四川彭山东南），留下参军周楚、孙盛看守辎重，自己亲率步兵直攻成都。同月，成汉将领李福袭击彭模，结果被孙盛等人击退；而桓温三战三胜，一直逼近成都。

首先我们进行分词，并把分词结果分别存在nlp_test1.txt, nlp_test3.txt和 nlp_test5.txt：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import jieba
jieba.suggest_freq('沙瑞金', True)
jieba.suggest_freq('易学习', True)
jieba.suggest_freq('王大路', True)
jieba.suggest_freq('京州', True)

#第一个文档分词#
with open('./nlp_test0.txt') as f:
    document = f.read()

    document_decode = document.decode('GBK')
    document_cut = jieba.cut(document_decode)
    #print ' '.join(document_cut)
    result = ' '.join(document_cut)
    result = result.encode('utf-8')
    with open('./nlp_test1.txt', 'w') as f2:
        f2.write(result)
f.close()
f2.close()

#第二个文档分词#
with open('./nlp_test2.txt') as f:
    document2 = f.read()

    document2_decode = document2.decode('GBK')
    document2_cut = jieba.cut(document2_decode)
    #print ' '.join(document2_cut)
    result = ' '.join(document2_cut)
    result = result.encode('utf-8')
    with open('./nlp_test3.txt', 'w') as f2:
        f2.write(result)
f.close()
f2.close()

#第三个文档分词#
jieba.suggest_freq('桓温', True)
with open('./nlp_test4.txt') as f:
    document3 = f.read()

    document3_decode = document3.decode('GBK')
    document3_cut = jieba.cut(document3_decode)
    #print ' '.join(document3_cut)
    result = ' '.join(document3_cut)
    result = result.encode('utf-8')
    with open('./nlp_test5.txt', 'w') as f3:
        f3.write(result)
f.close()
f3.close()
```

现在我们读入分好词的数据到内存备用，并打印分词结果观察：

```
with open('./nlp_test1.txt') as f3:  
    res1 = f3.read()  
print res1  
with open('./nlp_test3.txt') as f4:  
    res2 = f4.read()  
print res2  
with open('./nlp_test5.txt') as f5:  
    res3 = f5.read()  
print res3
```

打印出的分词结果如下：

沙瑞金 赞叹 易学习 的 胸怀 ， 是 金山 的 百姓 有福 ， 可是 这件 事对 李达康 的 触动 很大 。 易学习 又 回忆起 他们 三 人 分开 的 前一晚 ， 大家 一起 喝酒话别 ， 易学习 被 降职 到 道口 县当 县长 ， 王大路 下海经商 ， 李达康 连连 赔礼道歉 ， 觉得 对不起 大家 ， 他 最 对不起 的 是 王大路 ， 就 和 易学习 一起 给 王大路 凑 了 5 万块 钱 ， 王大路 自己 东挪西撮 了 5 万块 ， 开始 下海经商 。 没想到 后来 王大路 竟然 做得 风生水起 。 沙瑞金 觉得 他们 三 人 ， 在 困难 时期 还能 以沫 相助 ， 很 不容易 。

沙瑞金 向 毛娅 打听 他们 家 在 京都 的 别墅 ， 毛娅 笑着 说 ， 王大路 事业 有成 之后 ， 要 给 欧阳菁 和 她 公司 的 股权 ， 她们 没有 要 ， 王大路 就 在 京都 帝豪园 买了 三套 别墅 ， 可是 李达康 和 易学习 都 不要 ， 这些 房子 都在 王大路 的 名下 ， 欧阳菁 好像 去住 过 ， 毛娅 不想 去 ， 她 觉得 房子 太大 很 浪费 ， 自己 家住 得 就 很 踏实 。

347 年（永和三年）三月，桓温兵至彭模（今四川彭山东南），留下参军周楚、孙盛看守辎重，自己亲率步兵直攻成都。同月，成汉将领李福袭击彭模，结果被孙盛等人击退；而桓温三战三胜，一直逼近成都。

我们接着导入停用词表，这里的代码和[中文文本挖掘预处理流程总结](#)中一样，如果大家没有1208个的中文停用词表，可以到之前的这篇文章的链接里去下载。

```
#从文件导入停用词表  
stpwrdpath = "stop_words.txt"  
stpwrd_dic = open(stpwrdpath, 'rb')  
stpwrd_content = stpwrd_dic.read()  
#将停用词表转换为list  
stpwrdlst = stpwrd_content.splitlines()  
stpwrd_dic.close()
```

接着我们要把词转化为词频向量，注意由于LDA是基于词频统计的，因此一般不用TF-IDF来做文档特征。代码如下：

```
from sklearn.feature_extraction.text import CountVectorizer  
from sklearn.decomposition import LatentDirichletAllocation  
corpus = [res1,res2,res3]  
cntVector = CountVectorizer(stop_words=stpwrdlst)  
cntTf = cntVector.fit_transform(corpus)  
print cntTf
```

输出即为所有文档中各个词的词频向量。有了这个词频向量，我们就可以来做LDA主题模型了，由于我们只有三个文档，所以选择主题数 $K=2$ 。代码如下：

```
lda = LatentDirichletAllocation(n_topics=2,  
                                 learning_offset=50.,  
                                 random_state=0)  
docres = lda.fit_transform(cntTf)
```

通过fit_transform函数，我们就可以得到文档的主题模型分布在docres中。而主题词分布则在lda.components_中。我们将其打印出来：

```
print docres  
print lda.components_
```

文档主题的分布如下：

```
[[ 0.00950072  0.99049928]
 [ 0.0168786   0.9831214  ]
 [ 0.98429257  0.01570743]]
```

可见第一个和第二个文档较大概率属于主题2，则第三个文档属于主题1。

主题和词的分布如下：

```
[[ 1.32738199  1.24830645  0.90453117  0.7416939   0.78379936  0.89659305
  1.26874773  1.23261029  0.82094727  0.87788498  0.94980757  1.21509469
  0.64793292  0.89061203  1.00779152  0.70321998  1.04526968  1.30907884
  0.81932312  0.67798129  0.93434765  1.2937011   1.170592   0.70423093
  0.93400364  0.75617108  0.69258778  0.76780266  1.17923311  0.88663943
  1.2244191   0.88397724  0.74734167  1.20690264  0.73649036  1.1374004
  0.69576496  0.8041923   0.83229086  0.8625258   0.88495323  0.8207144
  1.66806345  0.85542475  0.71686887  0.84556777  1.25124491  0.76510471
  0.84978448  1.21600212  1.66496509  0.84963486  1.24645499  1.72519498
  1.23308705  0.97983681  0.77222879  0.8339811   0.85949947  0.73931864
  1.33412296  0.91591144  1.6722457   0.98800604  1.26042063  1.09455497
  1.24696097  0.81048961  0.79308036  0.95030603  0.83259407  1.19681066
  1.18562629  0.80911991  1.19239034  0.81864393  1.24837997  0.72322227
  1.23471832  0.89962384  0.7307045   1.39429334  1.22255041  0.98600185
  0.77407283  0.74372971  0.71807656  0.75693778  0.83817087  1.33723701
  0.79249005  0.82589143  0.72502086  1.14726838  0.83487136  0.79650741
  0.72292882  0.81856129]
 [ 0.72740212  0.73371879  1.64230568  1.5961744   1.70396534  1.04072318
  0.71245387  0.77316486  1.59584637  1.15108883  1.15939659  0.76124093
  1.34750239  1.21659215  1.10029347  1.20616038  1.56146506  0.80602695
  2.05479544  1.18041584  1.14597993  0.76459826  0.8218473   1.2367587
  1.44906497  1.19538763  1.35241035  1.21501862  0.7460776   1.61967022
  0.77892814  1.14830281  1.14293716  0.74425664  1.18887759  0.79427197
  1.15820484  1.26045121  1.69001421  1.17798335  1.12624327  1.12397988
  0.83866079  1.2040445   1.24788376  1.63296361  0.80850841  1.19119425
  1.1318814   0.80423837  0.74137153  1.21226307  0.67200183  0.78283995
  0.75366187  1.5062978   1.27081319  1.2373463   2.99243195  1.21178667
  0.66714016  2.17440219  0.73626368  1.60196863  0.71547934  1.94575151
  0.73691176  2.02892667  1.3528508   0.0655887   1.1460755   4.17528123
  0.74939365  1.23685079  0.76431961  1.17922085  0.70112531  1.14761871
  0.80877956  1.12307426  1.21107782  1.64947394  0.74983027  2.03800612
  1.21378076  1.21213961  1.23397206  1.16994431  1.07224768  0.75292945
  1.10391419  1.26932908  1.26207274  0.70943937  1.1236972   1.24175001
  1.27929042  1.19130408]]
```

在实际的应用中，我们需要对 K, α, η 进行调参。如果是"online"算法，则可能需要对"online"算法的一些参数做调整。这里只是给出了LDA主题模型从原始文档到实际LDA处理的过程。希望可以帮助到大家。

隐马尔科夫模型HMM (一) HMM模型

[隐马尔科夫模型HMM \(一\) HMM模型基础](#)

[隐马尔科夫模型HMM \(二\) 前向后向算法评估观察序列概率](#)

[隐马尔科夫模型HMM \(三\) 鲍姆-韦尔奇算法求解HMM参数](#)

[隐马尔科夫模型HMM \(四\) 维特比算法解码隐藏状态序列](#)

隐马尔科夫模型 (Hidden Markov Model, 以下简称HMM) 是比较经典的机器学习模型了，它在语言识别、自然语言处理、模式识别等领域得到广泛的应用。当然，随着目前深度学习的崛起，尤其是RNN、LSTM等神经网络序列模型的火热，HMM的地位有所下降。但是作为一个经典的模型，学习HMM的模型和对应算法，对我们解决问题建模的能力提高以及算法思路的拓展还是很好的。本文是HMM系列的第一篇，关注于HMM模型的基础。

1. 什么样的问题需要HMM模型

首先我们来看看什么样的问题解决可以用HMM模型。使用HMM模型时我们的问题一般有这两个特征：1) 我们的问题是基于序列的，比如时间序列，或者状态序列。2) 我们的问题中有两类数据，一类序列数据是可以观测到的，即观测序列；而另一类数据是不能观察到的，即隐藏状态序列，简称状态序列。

有了这两个特征，那么这个问题一般可以用HMM模型来尝试解决。这样的问题在实际生活中是很多的。比如：我现在在打字写博客，我在键盘上敲出来的一系列字符就是观测序列，而我实际想写的一段话就是隐藏序列，输入法的任务就是从敲入的一系列字符尽可能的猜测我要写的一段话，并把最可能的词语放在最前面让我选择，这就可以看做一个HMM模型了。再举一个，我在和你说话，我发出的一串连续的声音就是观测序列，而我实际要表达的一段话就是状态序列，你大脑的任务，就是从这一串连续的声音中判断出我最可能要表达的话的内容。

从这些例子中，我们可以发现，HMM模型可以无处不在。但是上面的描述还不精确，下面我们用精确的数学符号来表述我们的HMM模型。

2. HMM模型的定义

对于HMM模型，首先我们假设 Q 是所有可能的隐藏状态的集合， V 是所有可能的观测状态的集合，即：

$$Q = \{q_1, q_2, \dots, q_N\}, V = \{v_1, v_2, \dots, v_M\}$$

其中， N 是可能的隐藏状态数， M 是所有的可能的观察状态数。

对于一个长度为 T 的序列， I 对应的状态序列， O 是对应的观察序列，即：

$$I = \{i_1, i_2, \dots, i_T\}, O = \{o_1, o_2, \dots, o_T\}$$

其中，任意一个隐藏状态 $i_t \in Q$ ，任意一个观察状态 $o_t \in V$

HMM模型做了两个很重要的假设如下：

1) 齐次马尔科夫链假设。即任意时刻的隐藏状态只依赖于它前一个隐藏状态，这个我们在MCMC(二)马尔科夫链中有详细讲述。当然这样假设有点极端，因为很多时候我们的某一个隐藏状态不仅仅只依赖于前一个隐藏状态，可能是前两个或者是前三个。但是这样假设的好处就是模型简单，便于求解。如果在时刻 t 的隐藏状态是 $i_t = q_i$ ，在时刻 $t+1$ 的隐藏状态是 $i_{t+1} = q_j$ ，则从时刻 t 到时刻 $t+1$ 的HMM状态转移概率 a_{ij} 可以表示为：

$$a_{ij} = P(i_{t+1} = q_j | i_t = q_i)$$

这样 a_{ij} 可以组成马尔科夫链的状态转移矩阵 A ：

$$A = [a_{ij}]_{N \times N}$$

2) 观测独立性假设。即任意时刻的观察状态只仅仅依赖于当前时刻的隐藏状态，这也是一个为了简化模型的假设。如果在时刻 t 的隐藏状态是 $i_t = q_j$ ，而对应的观察状态为 $o_t = v_k$ ，则该时刻观察状态 v_k 在隐藏状态 q_j 下生成的概率为 $b_j(k)$ ，满足：

$$b_j(k) = P(o_t = v_k | i_t = q_j)$$

这样 $b_j(k)$ 可以组成观测状态生成的概率矩阵 B ：

$$B = [b_j(k)]_{N \times M}$$

除此之外，我们需要一组在时刻 $t = 1$ 的隐藏状态概率分布 Π :

$$\Pi = [\pi(i)]_N \text{ 其中 } \pi(i) = P(i_1 = q_i)$$

一个HMM模型，可以由隐藏状态初始概率分布 Π ，状态转移概率矩阵 A 和观测状态概率矩阵 B 决定。 Π, A 决定状态序列， B 决定观测序列。因此，HMM模型可以由一个三元组表示如下：

$$\lambda = (A, B, \Pi)$$

3. 一个HMM模型实例

下面我们用一个简单的实例来描述上面抽象出的HMM模型。这是一个盒子与球的模型，例子来源于李航的《统计学习方法》。

假设我们有3个盒子，每个盒子里都有红色和白色两种球，这三个盒子里球的数量分别是：

盒子	1	2	3
红球数	5	4	7
白球数	5	6	3

按照下面的方法从盒子里抽球，开始的时候，从第一个盒子抽球的概率是0.2，从第二个盒子抽球的概率是0.4，从第三个盒子抽球的概率是0.4。以这个概率抽一次球后，将球放回。然后从当前盒子转移到下一个盒子进行抽球。规则是：如果当前抽球的盒子是第一个盒子，则以0.5的概率仍然留在第一个盒子继续抽球，以0.2的概率去第二个盒子抽球，以0.3的概率去第三个盒子抽球。如果当前抽球的盒子是第二个盒子，则以0.5的概率仍然留在第二个盒子继续抽球，以0.3的概率去第一个盒子抽球，以0.2的概率去第三个盒子抽球。如果当前抽球的盒子是第三个盒子，则以0.5的概率仍然留在第三个盒子继续抽球，以0.2的概率去第一个盒子抽球，以0.3的概率去第二个盒子抽球。如此下去，直到重复三次，得到一个球的颜色的观测序列：

$$O = \{\text{红}, \text{白}, \text{红}\}$$

注意在这个过程中，观察者只能看到球的颜色序列，却不能看到球是从哪个盒子里取出的。

那么按照我们上一节HMM模型的定义，我们的观察集合是：

$$V = \{\text{红}, \text{白}\}, M = 2$$

我们的状态集合是：

$$Q = \{\text{盒子1}, \text{盒子2}, \text{盒子3}\}, N = 3$$

而观察序列和状态序列的长度为3.

初始状态分布为：

$$\Pi = (0.2, 0.4, 0.4)^T$$

状态转移概率分布矩阵为：

$$A = \begin{pmatrix} 0.5 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

观测状态概率矩阵为：

$$B = \begin{pmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \\ 0.7 & 0.3 \end{pmatrix}$$

4. HMM观测序列的生成

从上一节的例子，我们也可以抽象出HMM观测序列生成的过程。

输入的是HMM的模型 $\lambda = (A, B, \Pi)$, 观测序列的长度 T

输出是观测序列 $O = \{o_1, o_2, \dots, o_T\}$

生成的过程如下：

1) 根据初始状态概率分布 Π 生成隐藏状态 i_1

2) for t from 1 to T

- a. 按照隐藏状态 i_t 的观测状态分布 $b_{i_t}(k)$ 生成观察状态 o_t
- b. 按照隐藏状态 i_t 的状态转移概率分布 $a_{i_t, i_{t+1}}$ 产生隐藏状态 i_{t+1}

所有的 o_t 一起形成观测序列 $O = \{o_1, o_2, \dots, o_T\}$

5. HMM模型的三个基本问题

HMM模型一共有三个经典的问题需要解决：

1) 评估观察序列概率。即给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = \{o_1, o_2, \dots, o_T\}$ ，计算在模型 λ 下观测序列 O 出现的概率 $P(O|\lambda)$ 。这个问题的求解需要用到前向后向算法，我们在这个系列的第二篇会详细讲解。这个问题是HMM模型三个问题中最简单的。

2) 模型参数学习问题。即给定观测序列 $O = \{o_1, o_2, \dots, o_T\}$ ，估计模型 $\lambda = (A, B, \Pi)$ 的参数，使该模型下观测序列的条件概率 $P(O|\lambda)$ 最大。这个问题的求解需要用到基于EM算法的鲍姆-韦尔奇算法，我们在这个系列的第三篇会详细讲解。这个问题是HMM模型三个问题中最复杂的。

3) 预测问题，也称为解码问题。即给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = \{o_1, o_2, \dots, o_T\}$ ，求给定观测序列条件下，最可能出现的对应的状态序列，这个问题的求解需要用到基于动态规划的维特比算法，我们在这个系列的第四篇会详细讲解。这个问题是HMM模型三个问题中复杂度居中的算法。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率

[隐马尔科夫模型HMM（一）HMM模型](#)

[隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率](#)

[隐马尔科夫模型HMM（三）鲍姆-韦尔奇算法求解HMM参数](#)

[隐马尔科夫模型HMM（四）维特比算法解码隐藏状态序列](#)

在隐马尔科夫模型HMM（一）HMM模型中，我们讲到了HMM模型的基础知识和HMM的三个基本问题，本篇我们就关注于HMM第一个基本问题的解决方法，即已知模型和观测序列，求观测序列出现的概率。

1. 回顾HMM问题一：求观测序列的概率

首先我们回顾下HMM模型的问题一。这个问题是这样的。我们已知HMM模型的参数 $\lambda = (A, B, \Pi)$ 。其中 A 是隐藏状态转移概率的矩阵， B 是观测状态生成概率的矩阵， Π 是隐藏状态的初始概率分布。同时我们也已经得到了观测序列 $O = \{o_1, o_2, \dots, o_T\}$ ，现在我们要求观测序列 O 在模型 λ 下出现的条件概率 $P(O|\lambda)$ 。

乍一看，这个问题很简单。因为我们知道所有的隐藏状态之间的转移概率和所有从隐藏状态到观测状态生成概率，那么我们是可以暴力求解的。

我们可以列举出所有可能出现的长度为 T 的隐藏序列 $I = \{i_1, i_2, \dots, i_T\}$ ，分布求出这些隐藏序列与观测序列 $O = \{o_1, o_2, \dots, o_T\}$ 的联合概率分布 $P(O, I|\lambda)$ ，这样我们就可以很容易的求出边缘分布 $P(O|\lambda)$ 了。

具体暴力求解的方法是这样的：首先，任意一个隐藏序列 $I = \{i_1, i_2, \dots, i_T\}$ 出现的概率是：

$$P(I|\lambda) = \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \dots a_{i_{T-1} i_T}$$

对于固定的状态序列 $I = \{i_1, i_2, \dots, i_T\}$ ，我们要求的观察序列 $O = \{o_1, o_2, \dots, o_T\}$ 出现的概率是：

$$P(O|I, \lambda) = b_{i_1}(o_1) b_{i_2}(o_2) \dots b_{i_T}(o_T)$$

则 O 和 I 联合出现的概率是：

$$P(O, I|\lambda) = P(I|\lambda)P(O|I, \lambda) = \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) \dots a_{i_{T-1} i_T} b_{i_T}(o_T)$$

然后求边缘概率分布，即可得到观测序列 O 在模型 λ 下出现的条件概率 $P(O|\lambda)$ ：

$$P(O|\lambda) = \sum_I P(O, I|\lambda) = \sum_{i_1, i_2, \dots, i_T} \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) \dots a_{i_{T-1} i_T} b_{i_T}(o_T)$$

虽然上述方法有效，但是如果我们的隐藏状态数 N 非常多的那就麻烦了，此时我们预测状态有 N^T 种组合，算法的时间复杂度是 $O(TN^T)$ 阶的。因此对于一些隐藏状态数极少的模型，我们可以用暴力求解法来得到观测序列出现的概率，但是如果隐藏状态多，则上述算法太耗时，我们需要寻找其他简洁的算法。

前向后向算法就是来帮助我们在较低的时间复杂度情况下求解这个问题的。

2. 用前向算法求HMM观测序列的概率

前向后向算法是前向算法和后向算法的统称，这两个算法都可以用来求HMM观测序列的概率。我们先来看看前向算法是如何求解这个问题的。

前向算法本质上属于动态规划的算法，也就是我们要通过找到局部状态递推的公式，这样一步步的从子问题的最优解拓展到整个问题的最优解。

在前向算法中，通过定义“前向概率”来定义动态规划的这个局部状态。什么是前向概率呢，其实定义很简单：定义时刻 t 时隐藏状态为 q_i ，观测状态的序列为 o_1, o_2, \dots, o_t 的概率为前向概率。记为：

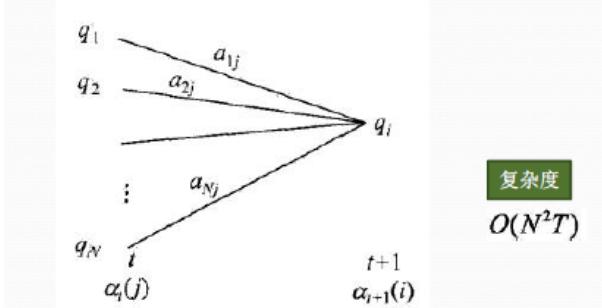
$$\alpha_t(i) = P(o_1, o_2, \dots, o_t, i_t = q_i | \lambda)$$

既然是动态规划，我们就要递推了，现在我们假设我们已经找到了在时刻 t 时各个隐藏状态的前向概率，现在我们需要递推出时刻 $t+1$ 时各个隐藏状态的前向概率。

从下图可以看出，我们可以基于时刻 t 时各个隐藏状态的前向概率，再乘以对应的状态转移概率，即 $\alpha_t(j)a_{ji}$ 就是在时刻 t 观测到 o_1, o_2, \dots, o_t ，并且时刻 t 隐藏状态 q_j ，时刻 $t+1$ 隐藏状态 q_i 的概率。如果将想下面所有的线对应的概率求和，

即 $\sum_{j=1}^N \alpha_t(j) a_{ji}$ 就是在时刻 t 观测到 o_1, o_2, \dots, o_t ，并且时刻 $t+1$ 隐藏状态 q_i 的概率。继续一步，由于观测状态 o_{t+1} 只依赖于 $t+1$ 时刻隐藏状态 q_i ，这样 $[\sum_{j=1}^N \alpha_t(j) a_{ji}] b_i(o_{t+1})$ 就是在在时刻 $t+1$ 观测到 $o_1, o_2, \dots, o_t, o_{t+1}$ ，并且时刻 $t+1$ 隐藏状态 q_i 的概率。而这个概率，恰恰就是时刻 $t+1$ 对应的隐藏状态 i 的前向概率，这样我们得到了前向概率的递推关系式如下：

$$\alpha_{t+1}(i) = \left[\sum_{j=1}^N \alpha_t(j) a_{ji} \right] b_i(o_{t+1})$$



我们的动态规划从时刻 1 开始，到时刻 T 结束，由于 $\alpha_T(i)$ 表示在时刻 T 观测序列为 o_1, o_2, \dots, o_T ，并且时刻 T 隐藏状态 q_i 的概率，我们只要将所有隐藏状态对应的概率相加，即 $\sum_{i=1}^N \alpha_T(i)$ 就得到了在时刻 T 观测序列为 o_1, o_2, \dots, o_T 的概率。

下面总结下前向算法。

输入：HMM 模型 $\lambda = (A, B, \Pi)$ ，观测序列 $O = (o_1, o_2, \dots, o_T)$

输出：观测序列概率 $P(O|\lambda)$

1) 计算时刻 1 的各个隐藏状态前向概率：

$$\alpha_1(i) = \pi_i b_i(o_1), i = 1, 2, \dots, N$$

2) 递推时刻 $2, 3, \dots, T$ 时刻的前向概率：

$$\alpha_{t+1}(i) = \left[\sum_{j=1}^N \alpha_t(j) a_{ji} \right] b_i(o_{t+1}), i = 1, 2, \dots, N$$

3) 计算最终结果：

$$P(O|\lambda) = \sum_{i=1}^N \alpha_T(i)$$

从递推公式可以看出，我们的算法时间复杂度是 $O(TN^2)$ ，比暴力解法的时间复杂度 $O(TN^T)$ 少了几个数量级。

3. HMM 前向算法求解实例

这里我们用隐马尔科夫模型 HMM (一) HMM 模型中盒子与球的例子来显示前向概率的计算。

我们的观察集合是：

$$V = \{\text{红}, \text{白}\}, M = 2$$

我们的状态集合是：

$$Q = \{\text{盒子1}, \text{盒子2}, \text{盒子3}\}, N = 3$$

而观察序列和状态序列的长度为 3。

初始状态分布为：

$$\Pi = (0.2, 0.4, 0.4)^T$$

状态转移概率分布矩阵为：

$$A = \begin{pmatrix} 0.5 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

观测状态概率矩阵为：

$$B = \begin{pmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \\ 0.7 & 0.3 \end{pmatrix}$$

球的颜色的观测序列：

$$O = \{\text{红}, \text{白}, \text{红}\}$$

按照我们上一节的前向算法。首先计算时刻1三个状态的前向概率：

时刻1是红色球，隐藏状态是盒子1的概率为：

$$\alpha_1(1) = \pi_1 b_1(o_1) = 0.2 \times 0.5 = 0.1$$

隐藏状态是盒子2的概率为：

$$\alpha_1(2) = \pi_2 b_2(o_1) = 0.4 \times 0.4 = 0.16$$

隐藏状态是盒子3的概率为：

$$\alpha_1(3) = \pi_3 b_3(o_1) = 0.4 \times 0.7 = 0.28$$

现在我们可以开始递推了，首先递推时刻2三个状态的前向概率：

时刻2是白色球，隐藏状态是盒子1的概率为：

$$\alpha_2(1) = \left[\sum_{i=1}^3 \alpha_1(i) a_{i1} \right] b_1(o_2) = [0.1 * 0.5 + 0.16 * 0.3 + 0.28 * 0.2] \times 0.5 = 0.077$$

隐藏状态是盒子2的概率为：

$$\alpha_2(2) = \left[\sum_{i=1}^3 \alpha_1(i) a_{i2} \right] b_2(o_2) = [0.1 * 0.2 + 0.16 * 0.5 + 0.28 * 0.3] \times 0.6 = 0.1104$$

隐藏状态是盒子3的概率为：

$$\alpha_2(3) = \left[\sum_{i=1}^3 \alpha_1(i) a_{i3} \right] b_3(o_2) = [0.1 * 0.3 + 0.16 * 0.2 + 0.28 * 0.5] \times 0.3 = 0.0606$$

继续递推，现在我们递推时刻3三个状态的前向概率：

时刻3是红色球，隐藏状态是盒子1的概率为：

$$\alpha_3(1) = \left[\sum_{i=1}^3 \alpha_2(i) a_{i1} \right] b_1(o_3) = [0.077 * 0.5 + 0.1104 * 0.3 + 0.0606 * 0.2] \times 0.5 = 0.04187$$

隐藏状态是盒子2的概率为：

$$\alpha_3(2) = \left[\sum_{i=1}^3 \alpha_2(i) a_{i2} \right] b_2(o_3) = [0.077 * 0.2 + 0.1104 * 0.5 + 0.0606 * 0.3] \times 0.4 = 0.03551$$

隐藏状态是盒子3的概率为：

$$\alpha_3(3) = \left[\sum_{i=1}^3 \alpha_2(i) a_{i3} \right] b_3(o_3) = [0.077 * 0.3 + 0.1104 * 0.2 + 0.0606 * 0.5] \times 0.7 = 0.05284$$

最终我们求出观测序列： $O = \{\text{红}, \text{白}, \text{红}\}$ 的概率为：

$$P(O|\lambda) = \sum_{i=1}^3 \alpha_3(i) = 0.13022$$

4. 用后向算法求HMM观测序列的概率

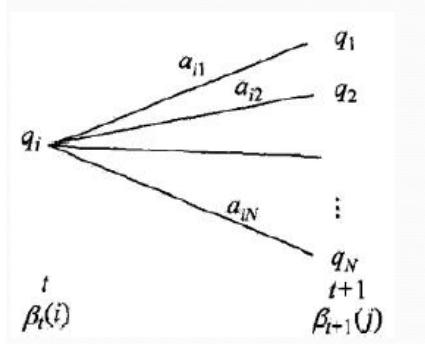
熟悉了用前向算法求HMM观测序列的概率，现在我们再来看看怎么用后向算法求HMM观测序列的概率。

后向算法和前向算法非常类似，都是用的动态规划，唯一的区别是选择的局部状态不同，后向算法用的是“后向概率”，那么后向概率是如何定义的呢？

定义时刻t时隐藏状态为 q_i ，从时刻 $t+1$ 到最后时刻T的观测状态的序列为 $o_{t+1}, o_{t+2}, \dots, o_T$ 的概率为后向概率。记为：

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, \dots, o_T | i_t = q_i, \lambda)$$

后向概率的动态规划递推公式和前向概率是相反的。现在我们假设我们已经找到了在时刻 $t+1$ 时各个隐藏状态的后向概率 $\beta_{t+1}(j)$ ，现在我们需要递推出时刻t时各个隐藏状态的后向概率。如下图，我们可以计算出观测状态的序列为 $o_{t+2}, o_{t+3}, \dots, o_T$ ，t时隐藏状态为 q_i ，时刻 $t+1$ 隐藏状态为 q_j 的概率为 $a_{ij} \beta_{t+1}(j)$ ，接着可以得到观测状态的序列为 $o_{t+1}, o_{t+2}, \dots, o_T$ ，t时隐藏状态为 q_i ，时刻 $t+1$ 隐藏状态为 q_j 的概率为 $a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$ ，则把下面所有线对应的概率加起来，我们可以得到观测状态的序列为 $o_{t+1}, o_{t+2}, \dots, o_T$ ，t时隐藏状态为 q_i 的概率为 $\sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$ ，这个概率即为时刻t的后向概率。



这样我们得到了后向概率的递推关系式如下：

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$$

现在我们总结下后向算法的流程,注意下和前向算法的相同点和不同点：

输入 : HMM模型 $\lambda = (A, B, \Pi)$, 观测序列 $O = (o_1, o_2, \dots, o_T)$

输出 : 观测序列概率 $P(O|\lambda)$

1) 初始化时刻 T 的各个隐藏状态后向概率 :

$$\beta_T(i) = 1, i = 1, 2, \dots, N$$

2) 递推时刻 $T-1, T-2, \dots, 1$ 时刻的后向概率 :

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), i = 1, 2, \dots, N$$

3) 计算最终结果 :

$$P(O|\lambda) = \sum_{i=1}^N \pi_i b_i(o_1) \beta_1(i)$$

此时我们的算法时间复杂度仍然是 $O(TN^2)$ 。

5. HMM常用概率的计算

利用前向概率和后向概率，我们可以计算出HMM中单个状态和两个状态的概率公式。

1) 给定模型 λ 和观测序列 O , 在时刻 t 处于状态 q_i 的概率记为:

$$\gamma_t(i) = P(i_t = q_i | O, \lambda) = \frac{P(i_t = q_i, O | \lambda)}{P(O | \lambda)}$$

利用前向概率和后向概率的定义可知：

$$P(i_t = q_i, O | \lambda) = \alpha_t(i) \beta_t(i)$$

于是我们得到：

$$\gamma_t(i) = \frac{\alpha_t(i) \beta_t(i)}{\sum_{j=1}^N \alpha_t(j) \beta_t(j)}$$

2) 给定模型 λ 和观测序列 O , 在时刻 t 处于状态 q_i , 且时刻 $t+1$ 处于状态 q_j 的概率记为:

$$\xi_t(i, j) = P(i_t = q_i, i_{t+1} = q_j | O, \lambda) = \frac{P(i_t = q_i, i_{t+1} = q_j, O | \lambda)}{P(O | \lambda)}$$

而 $P(i_t = q_i, i_{t+1} = q_j, O | \lambda)$ 可以由前向后向概率来表示为:

$$P(i_t = q_i, i_{t+1} = q_j, O | \lambda) = \alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$$

从而最终我们得到 $\xi_t(i, j)$ 的表达式如下：

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_{r=1}^N \sum_{s=1}^N \alpha_t(r) a_{rs} b_s(o_{t+1}) \beta_{t+1}(s)}$$

3) 将 $\gamma_t(i)$ 和 $\xi_t(i, j)$ 在各个时刻 t 求和，可以得到：

$$\text{在观测序列 } O \text{ 下状态 } i \text{ 出现的期望值 } \sum_{t=1}^T \gamma_t(i)$$

在观测序列 O 下由状态 i 转移的期望值 $\sum_{t=1}^{T-1} \gamma_t(i)$

在观测序列 O 下由状态 i 转移到状态 j 的期望值 $\sum_{t=1}^{T-1} \xi_t(i, j)$

上面这些常用的概率值在求解HMM问题二，即求解HMM模型参数的时候需要用到。我们在这个系列的第三篇来讨论求解HMM参数的问题和解法。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

隐马尔科夫模型HMM (三) 鲍姆-韦尔奇算法求解HMM参数

[隐马尔科夫模型HMM \(一\) HMM模型](#)

[隐马尔科夫模型HMM \(二\) 前向后向算法评估观察序列概率](#)

[隐马尔科夫模型HMM \(三\) 鲍姆-韦尔奇算法求解HMM参数](#)

[隐马尔科夫模型HMM \(四\) 维特比算法解码隐藏状态序列](#)

在本篇我们会讨论HMM模型参数求解的问题，这个问题在HMM三个问题里算是最复杂的。在研究这个问题之前，建议先阅读这个系列的前两篇以熟悉HMM模型和HMM的前向后向算法，以及EM算法原理总结，这些在本篇里会用到。在李航的《统计学习方法》中，这个算法的讲解只考虑了单个观测序列的求解，因此无法用于实际多样本观测序列的模型求解，本文关注于如何使用多个观测序列来求解HMM模型参数。

1. HMM模型参数求解概述

HMM模型参数求解根据已知的条件可以分为两种情况。

第一种情况较为简单，就是我们已知 D 个长度为 T 的观测序列和对应的隐藏状态序列，即 $\{(O_1, I_1), (O_2, I_2), \dots, (O_D, I_D)\}$ 是已知的，此时我们可以很容易的用最大似然来求解模型参数。

假设样本从隐藏状态 q_i 转移到 q_j 的频率计数是 A_{ij} ，那么状态转移矩阵求得为：

$$A = [a_{ij}], \text{ 其中 } a_{ij} = \frac{A_{ij}}{\sum_{s=1}^N A_{is}}$$

假设样本隐藏状态为 q_j 且观测状态为 v_k 的频率计数是 B_{jk} ，那么观测状态概率矩阵为：

$$B = [b_j(k)], \text{ 其中 } b_j(k) = \frac{B_{jk}}{\sum_{s=1}^M B_{js}}$$

假设所有样本中初始隐藏状态为 q_i 的频率计数为 $C(i)$ ，那么初始概率分布为：

$$\Pi = \pi(i) = \frac{C(i)}{\sum_{s=1}^N C(s)}$$

可见第一种情况下求解模型还是很简单的。但是在很多时候，我们无法得到HMM样本观察序列对应的隐藏序列，只有 D 个长度为 T 的观测序列，即 $\{(O_1), (O_2), \dots, (O_D)\}$ 是已知的，此时我们能不能求出合适的HMM模型参数呢？这就是我们的第二种情况，也是我们本文要讨论的重点。它的解法最常用的是鲍姆-韦尔奇算法，其实就是基于EM算法的求解，只不过鲍姆-韦尔奇算法出现的时代，EM算法还没有被抽象出来，所以我们本文还是说鲍姆-韦尔奇算法。

2. 鲍姆-韦尔奇算法原理

鲍姆-韦尔奇算法原理既然使用的就是EM算法的原理，那么我们需要在E步求出联合分布 $P(O, I | \lambda)$ 基于条件概率 $P(I | O, \bar{\lambda})$ 的期望，其中 $\bar{\lambda}$ 为当前的模型参数，然后再M步最大化这个期望，得到更新的模型参数 λ 。接着不停的进行EM迭代，直到模型参数的值收敛为止。

首先来看看E步，当前模型参数为 $\bar{\lambda}$ ，联合分布 $P(O, I | \lambda)$ 基于条件概率 $P(I | O, \bar{\lambda})$ 的期望表达式为：

$$L(\lambda, \bar{\lambda}) = \sum_{d=1}^D \sum_I P(I | O, \bar{\lambda}) \log P(O, I | \lambda)$$

在M步，我们极大化上式，然后得到更新后的模型参数如下：

$$\bar{\lambda} = \arg \max_{\lambda} \sum_{d=1}^D \sum_I P(I | O, \bar{\lambda}) \log P(O, I | \lambda)$$

通过不断的E步和M步的迭代，直到 $\bar{\lambda}$ 收敛。下面我们来看看鲍姆-韦尔奇算法的推导过程。

3. 鲍姆-韦尔奇算法的推导

我们的训练数据为 $\{(O_1, I_1), (O_2, I_2), \dots, (O_D, I_D)\}$ ，其中任意一个观测序列 $O_d = \{o_1^{(d)}, o_2^{(d)}, \dots, o_T^{(d)}\}$ ，其对应的未知的隐藏状态序列表示为： $O_d = \{i_1^{(d)}, i_2^{(d)}, \dots, i_T^{(d)}\}$

首先看鲍姆-韦尔奇算法的E步，我们需要先计算联合分布 $P(O, I | \lambda)$ 的表达式如下：

$$P(O, I | \lambda) = \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) \dots a_{i_{T-1} i_T} b_{i_T}(o_T)$$

我们的E步得到的期望表达式为：

$$L(\lambda, \bar{\lambda}) = \sum_{d=1}^D \sum_I P(I | O, \bar{\lambda}) \log P(O, I | \lambda)$$

在M步我们要极大化上式。由于 $P(I | O, \bar{\lambda}) = P(I, O | \bar{\lambda}) / P(O | \bar{\lambda})$ ，而 $P(O | \bar{\lambda})$ 是常数，因此我们要极大化的式子等价于：

$$\bar{\lambda} = \arg \max_{\lambda} \sum_{d=1}^D \sum_I P(O, I | \bar{\lambda}) \log P(O, I | \lambda)$$

我们将上面 $P(O, I | \lambda)$ 的表达式带入我们的极大化式子，得到的表达式如下：

$$\bar{\lambda} = \arg \max_{\lambda} \sum_{d=1}^D \sum_I P(O, I | \bar{\lambda}) (\log \pi_{i_1} + \sum_{t=1}^{T-1} \log a_{i_t} a_{i_{t+1}} + \sum_{t=1}^T b_{i_t}(o_t))$$

我们的隐藏模型参数 $\lambda = (A, B, \Pi)$ ，因此下面我们只需要对上式分别对 A, B, Π 求导即可得到我们更新的模型参数 $\bar{\lambda}$

首先我们看看对模型参数 Π 的求导。由于 Π 只在上式中括号里的第一部分出现，因此我们对于 Π 的极大化式子为：

$$\bar{\pi}_i = \arg \max_{\pi_{i_1}} \sum_{d=1}^D \sum_I P(O, I | \bar{\lambda}) \log \pi_{i_1} = \arg \max_{\pi_i} \sum_{d=1}^D \sum_{i=1}^N P(O, i_1^{(d)} = i | \bar{\lambda}) \log \pi_i$$

由于 π_i 还满足 $\sum_{i=1}^N \pi_i = 1$ ，因此根据拉格朗日乘法，我们得到 π_i 要极大化的拉格朗日函数为：

$$\arg \max_{\pi_i} \sum_{d=1}^D \sum_{i=1}^N P(O, i_1^{(d)} = i | \bar{\lambda}) \log \pi_i + \gamma (\sum_{i=1}^N \pi_i - 1)$$

其中， γ 为拉格朗日系数。上式对 π_i 求偏导数并令结果为0，我们得到：

$$\sum_{d=1}^D P(O, i_1^{(d)} = i | \bar{\lambda}) + \gamma \pi_i = 0$$

令 i 分别等于从1到 N ，从上式可以得到 N 个式子，对这 N 个式子求和可得：

$$\sum_{d=1}^D P(O | \bar{\lambda}) + \gamma = 0$$

从上两式消去 γ ，得到 π_i 的表达式为：

$$\pi_i = \frac{\sum_{d=1}^D P(O, i_1^{(d)} = i | \bar{\lambda})}{\sum_{d=1}^D P(O | \bar{\lambda})} = \frac{\sum_{d=1}^D P(O, i_1^{(d)} = i | \bar{\lambda})}{D} = \frac{\sum_{d=1}^D P(i_1^{(d)} = i | O, \bar{\lambda})}{D} = \frac{\sum_{d=1}^D P(i_1^{(d)} = i | O^{(d)}, \bar{\lambda})}{D}$$

利用我们在隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率里第二节中前向概率的定义可得：

$$P(i_1^{(d)} = i | O^{(d)}, \bar{\lambda}) = \gamma_1^{(d)}(i)$$

因此最终我们在M步 π_i 的迭代公式为：

$$\pi_i = \frac{\sum_{d=1}^D \gamma_1^{(d)}(i)}{D}$$

现在我们来看看 A 的迭代公式求法。方法和 Π 的类似。由于 A 只在最大化函数式中括号里的第二部分出现，而这部分式子可以整理为：

$$\sum_{d=1}^D \sum_I \sum_{t=1}^{T-1} P(O, I | \bar{\lambda}) \log a_{i_t} a_{i_{t+1}} = \sum_{d=1}^D \sum_{i=1}^N \sum_{j=1}^N \sum_{t=1}^{T-1} P(O, i_t^{(d)} = i, i_{t+1}^{(d)} = j | \bar{\lambda}) \log a_{ij}$$

由于 a_{ij} 还满足 $\sum_{j=1}^N a_{ij} = 1$ 。和求解 π_i 类似，我们可以用拉格朗日乘法并对 a_{ij} 求导，并令结果为0，可以得到 a_{ij} 的迭代表达式为：

$$a_{ij} = \frac{\sum_{d=1}^D \sum_{t=1}^{T-1} P(O^{(d)}, i_t^{(d)} = i, i_{t+1}^{(d)} = j | \bar{\lambda})}{\sum_{d=1}^D \sum_{t=1}^{T-1} P(O^{(d)}, i_t^{(d)} = i | \bar{\lambda})}$$

利用隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率里第二节中前向概率的定义和第五节 $\xi_t(i, j)$ 的定义可得我们在M步 a_{ij} 的迭代公式为：

$$a_{ij} = \frac{\sum_{d=1}^D \sum_{t=1}^{T-1} \xi_t^{(d)}(i, j)}{\sum_{d=1}^D \sum_{t=1}^{T-1} \gamma_t^{(d)}(i)}$$

现在我们来看看 B 的迭代公式求法。方法和 Π 的类似。由于 B 只在最大化函数式中括号里的第三部分出现，而这部分式子可以整理为：

$$\sum_{d=1}^D \sum_{I} \sum_{t=1}^T P(O, I | \bar{\lambda}) \log b_{i_t}(o_t) = \sum_{d=1}^D \sum_{j=1}^N \sum_{t=1}^T P(O, i_t^{(d)} = j | \bar{\lambda}) \log b_j(o_t)$$

由于 $b_j(o_t)$ 还满足 $\sum_{k=1}^M b_j(o_t = v_k) = 1$ 。和求解 π_i 类似，我们可以用拉格朗日乘法并对 $b_j(k)$ 求导，并令结果为0，得到 $b_j(k)$ 的迭代表达式为：

$$b_j(k) = \frac{\sum_{d=1}^D \sum_{t=1}^T P(O, i_t^{(d)} = j | \bar{\lambda}) I(o_t^{(d)} = v_k)}{\sum_{d=1}^D \sum_{t=1}^T P(O, i_t^{(d)} = j | \bar{\lambda})}$$

其中 $I(o_t^{(d)} = v_k)$ 当且仅当 $o_t^{(d)} = v_k$ 时为1，否则为0。利用隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率里第二节中前向概率的定义可得 $b_j(o_t)$ 的最终表达式为：

$$b_j(k) = \frac{\sum_{d=1}^D \sum_{t=1, o_t^{(d)}=v_k}^T \gamma_t^{(d)}(i)}{\sum_{d=1}^D \sum_{t=1}^T \gamma_t^{(d)}(i)}$$

有了 $\pi_i, a_{ij}, b_j(k)$ 的迭代公式，我们就可以迭代求解HMM模型参数了。

4. 鲍姆-韦尔奇算法流程总结

这里我们概括总结下鲍姆-韦尔奇算法的流程。

输入： D 个观测序列样本 $\{(O_1), (O_2), \dots, (O_D)\}$

输出：HMM模型参数

1) 随机初始化所有的 $\pi_i, a_{ij}, b_j(k)$

2) 对于每个样本 $d = 1, 2, \dots, D$ ，用前向后向算法计算 $\gamma_t^{(d)}(i), \xi_t^{(d)}(i, j), t = 1, 2, \dots, T$

3) 更新模型参数：

$$\begin{aligned} \pi_i &= \frac{\sum_{d=1}^D \gamma_1^{(d)}(i)}{D} \\ a_{ij} &= \frac{\sum_{d=1}^D \sum_{t=1}^{T-1} \xi_t^{(d)}(i, j)}{\sum_{d=1}^D \sum_{t=1}^{T-1} \gamma_t^{(d)}(i)} \\ b_j(k) &= \frac{\sum_{d=1}^D \sum_{t=1, o_t^{(d)}=v_k}^T \gamma_t^{(d)}(i)}{\sum_{d=1}^D \sum_{t=1}^T \gamma_t^{(d)}(i)} \end{aligned}$$

4) 如果 $\pi_i, a_{ij}, b_j(k)$ 的值已经收敛，则算法结束，否则回到第2)步继续迭代。

以上就是鲍姆-韦尔奇算法的整个过程。

隐马尔科夫模型HMM（四）维特比算法解码隐藏状态序列

[隐马尔科夫模型HMM（一）HMM模型](#)

[隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率](#)

[隐马尔科夫模型HMM（三）鲍姆-韦尔奇算法求解HMM参数](#)

[隐马尔科夫模型HMM（四）维特比算法解码隐藏状态序列](#)

在本篇我们会讨论HMM模型最后一个问题是求解，即给定模型和观测序列，求给定观测序列条件下，最可能出现的对应的隐藏状态序列。在阅读本篇前，建议先阅读这个系列的第一篇以熟悉HMM模型。

HMM模型的解码问题最常用的算法是维特比算法，当然也有其他的算法可以求解这个问题。同时维特比算法是一个通用的求序列最短路径的动态规划算法，也可以用于很多其他问题，比如之前讲到的[文本挖掘的分词原理](#)中我们讲到了单独用维特比算法来做分词。

本文关注于用维特比算法来解码HMM的最可能隐藏状态序列。

1. HMM最可能隐藏状态序列求解概述

在HMM模型的解码问题中，给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = \{o_1, o_2, \dots, o_T\}$ ，求给定观测序列O条件下，最可能出现的对应的状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$ ，即 $P(I^* | O)$ 要最大化。

一个可能的近似解法是求出观测序列O在每个时刻t最可能的隐藏状态 i_t^* 然后得到一个近似的隐藏状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$ 。要这样近似求解不难，利用[隐马尔科夫模型HMM（二）前向后向算法评估观察序列概率](#)中第五节的定义：在给定模型 λ 和观测序列O时，在时刻t处于状态 q_i 的概率是 $\gamma_t(i)$ ，这个概率可以通过HMM的前向算法与后向算法计算。这样我们有：

$$i_t^* = \arg \max_{1 \leq i \leq N} [\gamma_t(i)], t = 1, 2, \dots, T$$

近似算法很简单，但是却不能保证预测的状态序列是整体是最可能的状态序列，因为预测的状态序列中某些相邻的隐藏状态可能存在转移概率为0的情况。

而维特比算法可以将HMM的状态序列作为一个整体来考虑，避免近似算法的问题，下面我们来看看维特比算法进行HMM解码的方法。

2. 维特比算法概述

维特比算法是一个通用的解码算法，是基于动态规划的求序列最短路径的方法。[文本挖掘的分词原理](#)中我们已经讲到了维特比算法的一些细节。

既然是动态规划算法，那么就需要找到合适的局部状态，以及局部状态的递推公式。在HMM中，维特比算法定义了两个局部状态用于递推。

第一个局部状态是在时刻t隐藏状态为*i*所有可能的状态转移路径 i_1, i_2, \dots, i_t 中的概率最大值。记为 $\delta_t(i)$ ：

$$\delta_t(i) = \max_{i_1, i_2, \dots, i_{t-1}} P(i_t = i, i_1, i_2, \dots, i_{t-1}, o_t, o_{t-1}, \dots, o_1 | \lambda), i = 1, 2, \dots, N$$

由 $\delta_t(i)$ 的定义可以得到 δ 的递推表达式：

$$\delta_{t+1}(i) = \max_{i_1, i_2, \dots, i_t} P(i_{t+1} = i, i_1, i_2, \dots, i_t, o_{t+1}, o_t, \dots, o_1 | \lambda) \quad (1)$$

$$= \max_{1 \leq j \leq N} [\delta_t(j)a_{ji}] b_i(o_{t+1}) \quad (2)$$

第二个局部状态由第一个局部状态递推得到。我们定义在时刻t隐藏状态为*i*的所有单个状态转移路径 $(i_1, i_2, \dots, i_{t-1}, i)$ 中概率最大的转移路径中第 $t - 1$ 个节点的隐藏状态为 $\Psi_t(i)$ ，其递推表达式可以表示为：

$$\Psi_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j)a_{ji}]$$

有了这两个局部状态，我们就可以从时刻0一直递推到时刻T，然后利用 $\Psi_t(i)$ 记录的前一个最可能的状态节点回溯，直到找到最优的隐藏状态序列。

3. 维特比算法流程总结

现在我们来总结下维特比算法的流程：

输入：HMM模型 $\lambda = (A, B, \Pi)$ ，观测序列 $O = (o_1, o_2, \dots, o_T)$

输出：最有可能的隐藏状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$

1) 初始化局部状态：

$$\delta_1(i) = \pi_i b_i(o_1), i = 1, 2, \dots, N$$

$$\Psi_1(i) = 0, i = 1, 2, \dots, N$$

2) 进行动态规划递推时刻 $t = 2, 3, \dots, T$ 时刻的局部状态：

$$\delta_t(i) = \max_{1 \leq j \leq N} [\delta_{t-1}(j)a_{ji}]b_i(o_t), i = 1, 2, \dots, N$$

$$\Psi_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j)a_{ji}], i = 1, 2, \dots, N$$

3) 计算时刻 T 最大的 $\delta_T(i)$ ，即为最可能隐藏状态序列出现的概率。计算时刻 T 最大的 $\Psi_T(i)$ ，即为时刻 T 最可能的隐藏状态。

$$P^* = \max_{1 \leq j \leq N} \delta_T(j)$$

$$i_T^* = \arg \max_{1 \leq j \leq N} [\delta_T(j)]$$

4) 利用局部状态 $\Psi(i)$ 开始回溯。对于 $t = T-1, T-2, \dots, 1$ ：

$$i_t^* = \Psi_{t+1}(i_{t+1}^*)$$

最终得到最有可能的隐藏状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$

4. HMM维特比算法求解实例

下面我们仍然用隐马尔科夫模型HMM（一）HMM模型中盒子与球的例子来看看HMM维特比算法求解。

我们的观察集合是：

$$V = \{\text{红}, \text{白}\}, M = 2$$

我们的状态集合是：

$$Q = \{\text{盒子1}, \text{盒子2}, \text{盒子3}\}, N = 3$$

而观察序列和状态序列的长度为3.

初始状态分布为：

$$\Pi = (0.2, 0.4, 0.4)^T$$

状态转移概率分布矩阵为：

$$A = \begin{pmatrix} 0.5 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

观测状态概率矩阵为：

$$B = \begin{pmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \\ 0.7 & 0.3 \end{pmatrix}$$

球的颜色的观测序列：

$$O = \{\text{红}, \text{白}, \text{红}\}$$

按照我们上一节的维特比算法，首先需要得到三个隐藏状态在时刻1时对应的各自两个局部状态，此时观测状态为1：

$$\delta_1(1) = \pi_1 b_1(o_1) = 0.2 \times 0.5 = 0.1$$

$$\delta_1(2) = \pi_2 b_2(o_1) = 0.4 \times 0.4 = 0.16$$

$$\delta_1(3) = \pi_3 b_3(o_1) = 0.4 \times 0.7 = 0.28$$

$$\Psi_1(1) = \Psi_1(2) = \Psi_1(3) = 0$$

现在开始递推三个隐藏状态在时刻2时对应的各自两个局部状态，此时观测状态为2：

$$\delta_2(1) = \max_{1 \leq j \leq 3} [\delta_1(j)a_{j1}]b_1(o_2) = \max_{1 \leq j \leq 3} [0.1 \times 0.5, 0.16 \times 0.3, 0.28 \times 0.2] \times 0.5 = 0.028$$

$$\Psi_2(1) = 3$$

$$\delta_2(2) = \max_{1 \leq j \leq 3} [\delta_1(j)a_{j2}]b_2(o_2) = \max_{1 \leq j \leq 3} [0.1 \times 0.2, 0.16 \times 0.5, 0.28 \times 0.3] \times 0.6 = 0.0504$$

$$\Psi_2(2) = 3$$

$$\delta_2(3) = \max_{1 \leq j \leq 3} [\delta_1(j)a_{j3}]b_3(o_2) = \max_{1 \leq j \leq 3} [0.1 \times 0.3, 0.16 \times 0.2, 0.28 \times 0.5] \times 0.3 = 0.042$$

$$\Psi_2(3) = 3$$

继续递推三个隐藏状态在时刻3时对应的各自两个局部状态，此时观测状态为1：

$$\delta_3(1) = \max_{1 \leq j \leq 3} [\delta_2(j)a_{j1}]b_1(o_3) = \max_{1 \leq j \leq 3} [0.028 \times 0.5, 0.0504 \times 0.3, 0.042 \times 0.2] \times 0.5 = 0.00756$$

$$\Psi_3(1) = 2$$

$$\delta_3(2) = \max_{1 \leq j \leq 3} [\delta_2(j)a_{j2}]b_2(o_3) = \max_{1 \leq j \leq 3} [0.028 \times 0.2, 0.0504 \times 0.5, 0.042 \times 0.3] \times 0.4 = 0.01008$$

$$\Psi_3(2) = 2$$

$$\delta_3(3) = \max_{1 \leq j \leq 3} [\delta_2(j)a_{j3}]b_3(o_3) = \max_{1 \leq j \leq 3} [0.028 \times 0.3, 0.0504 \times 0.2, 0.042 \times 0.5] \times 0.7 = 0.0147$$

$$\Psi_3(3) = 3$$

此时已经到最后的时刻，我们开始准备回溯。此时最大概率为 $\delta_3(3)$ ，从而得到 $i_3^* = 3$

由于 $\Psi_3(3) = 3$ ，所以 $i_2^* = 3$ ，而又由于 $\Psi_2(3) = 3$ ，所以 $i_1^* = 3$ 。从而得到最终的最可能的隐藏状态序列为：
(3, 3, 3)

5. HMM模型维特比算法总结

如果大家看过之前写的文本挖掘的分词原理中的维特比算法，就会发现这两篇之中的维特比算法稍有不同。主要原因是在中文分词时，我们没有观察状态和隐藏状态的区别，只有一种状态。但是维特比算法的核心是定义动态规划的局部状态与局部递推公式，这一点在中文分词维特比算法和HMM的维特比算法是相同的，也是维特比算法的精华所在。

维特比算法也是寻找序列最短路径的一个通用方法，和dijkstra算法有些类似，但是dijkstra算法并没有使用动态规划，而是贪心算法。同时维特比算法仅仅局限于求序列最短路径，而dijkstra算法是通用的求最短路径的方法。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

用hmmlearn学习隐马尔科夫模型HMM

在之前的HMM系列中，我们对隐马尔科夫模型HMM的原理以及三个问题的求解方法做了总结。本文我们就从实践的角度用Python的hmmlearn库来学习HMM的使用。关于hmmlearn的更多资料在[官方文档](#)有介绍。

1. hmmlearn概述

hmmlearn安装很简单，“`pip install hmmlearn`”即可完成。

hmmlearn实现了三种HMM模型类，按照观测状态是连续状态还是离散状态，可以分为两类。GaussianHMM和GMMHMM是连续观测状态的HMM模型，而MultinomialHMM是离散观测状态的模型，也是我们在HMM原理系列篇里面使用的模型。

对于MultinomialHMM的模型，使用比较简单，“`startprob_`”参数对应我们的隐藏状态初始分布 Π ，“`transmat_`”对应我们的状态转移矩阵 A ，“`emissionprob_`”对应我们的观测状态概率矩阵 B 。

对于连续观测状态的HMM模型，GaussianHMM类假设观测状态符合高斯分布，而GMMHMM类则假设观测状态符合混合高斯分布。一般情况下我们使用GaussianHMM即高斯分布的观测状态即可。以下对于连续观测状态的HMM模型，我们只讨论GaussianHMM类。

在GaussianHMM类中，“`startprob_`”参数对应我们的隐藏状态初始分布 Π ，“`transmat_`”对应我们的状态转移矩阵 A ，比较特殊的是观测状态概率的表示方法，此时由于观测状态是连续值，我们无法像MultinomialHMM一样直接给出矩阵 B 。而是采用给出各个隐藏状态对应的观测状态高斯分布的概率密度函数的参数。

如果观测序列是一维的，则观测状态的概率密度函数是一维的普通高斯分布。如果观测序列是 N 维的，则隐藏状态对应的观测状态的概率密度函数是 N 维高斯分布。高斯分布的概率密度函数参数可以用 μ 表示高斯分布的期望向量， Σ 表示高斯分布的协方差矩阵。在GaussianHMM类中，“means”用来表示各个隐藏状态对应的高斯分布期望向量 μ 形成的矩阵，而“covars”用来表示各个隐藏状态对应的高斯分布协方差矩阵 Σ 形成的三维张量。

2. MultinomialHMM实例

下面我们用我们在HMM系列原理篇中的例子来使用MultinomialHMM跑一遍。

首先建立HMM的模型：

```
import numpy as np
from hmmlearn import hmm

states = ["box 1", "box 2", "box3"]
n_states = len(states)

observations = ["red", "white"]
n_observations = len(observations)

start_probability = np.array([0.2, 0.4, 0.4])

transition_probability = np.array([
    [0.5, 0.2, 0.3],
    [0.3, 0.5, 0.2],
    [0.2, 0.3, 0.5]
])

emission_probability = np.array([
    [0.5, 0.5],
    [0.4, 0.6],
    [0.7, 0.3]
])

model = hmm.MultinomialHMM(n_components=n_states)
```

```
model.startprob_=start_probability  
model.transmat_=transition_probability  
model.emissionprob_=emission_probability
```



现在我们来跑一跑HMM问题三维特比算法的解码过程，使用和原理篇一样的观测序列来解码，代码如下：

```
seen = np.array([[0,1,0]]).T  
logprob, box = model.decode(seen, algorithm="viterbi")  
print("The ball picked:", ", ".join(map(lambda x: observations[x], seen)))  
print("The hidden box", ", ".join(map(lambda x: states[x], box)))
```

输出结果如下：

```
('The ball picked:', 'red, white, red')  
(The hidden box', 'box3, box3, box3')
```

可以看出，结果和我们原理篇中的手动计算的结果是一样的。

也可以使用predict函数，结果也是一样的，代码如下：

```
box2 = model.predict(seen)  
print("The ball picked:", ", ".join(map(lambda x: observations[x], seen)))  
print("The hidden box", ", ".join(map(lambda x: states[x], box2)))
```

大家可以跑一下，看看结果是否和decode函数相同。

现在我们再来看看求HMM问题一的观测序列的概率的问题，代码如下：

```
print model.score(seen)
```

输出结果是：

```
-2.03854530992
```

要注意的是score函数返回的是以自然对数为底的对数概率值，我们在HMM问题一中手动计算的结果是未取对数的原始概率是0.13022。对比一下：

$$\ln 0.13022 \approx -2.0385$$

现在我们再看看HMM问题二，求解模型参数的问题。由于鲍姆-韦尔奇算法是基于EM算法的近似算法，所以我们需要多跑几次，比如下面我们跑三次，选择一个比较优的模型参数，代码如下：

```
import numpy as np  
from hmmlearn import hmm  
  
states = ["box 1", "box 2", "box3"]  
n_states = len(states)  
  
observations = ["red", "white"]  
n_observations = len(observations)  
model2 = hmm.MultinomialHMM(n_components=n_states, n_iter=20, tol=0.01)  
X2 = np.array([[0,1,0,1],[0,0,0,1],[1,0,1,1]])  
model2.fit(X2)  
print model2.startprob_  
print model2.transmat_  
print model2.emissionprob_  
print model2.score(X2)  
model2.fit(X2)  
print model2.startprob_  
print model2.transmat_  
print model2.emissionprob_  
print model2.score(X2)  
model2.fit(X2)  
print model2.startprob_  
print model2.transmat_  
print model2.emissionprob_  
print model2.score(X2)
```



结果这里就略去了，最终我们会选择分数最高的模型参数。

以上就是用MultinomialHMM解决HMM模型三个问题的方法。

3. GaussianHMM实例

下面我们再给一个GaussianHMM的实例，这个实例中，我们的观测状态是二维的，而隐藏状态有4个。因此我们的“means”参数是 4×2 的矩阵，而“covars”参数是 $4 \times 2 \times 2$ 的张量。

建立模型如下：

```
startprob = np.array([0.6, 0.3, 0.1, 0.0])
# The transition matrix, note that there are no transitions possible
# between component 1 and 3
transmat = np.array([[0.7, 0.2, 0.0, 0.1],
                     [0.3, 0.5, 0.2, 0.0],
                     [0.0, 0.3, 0.5, 0.2],
                     [0.2, 0.0, 0.2, 0.6]])
# The means of each component
means = np.array([[0.0, 0.0],
                  [0.0, 11.0],
                  [9.0, 10.0],
                  [11.0, -1.0]])
# The covariance of each component
covars = .5 * np.tile(np.identity(2), (4, 1, 1))

# Build an HMM instance and set parameters
model3 = hmm.GaussianHMM(n_components=4, covariance_type="full")

# Instead of fitting it from the data, we directly set the estimated
# parameters, the means and covariance of the components
model3.startprob_ = startprob
model3.transmat_ = transmat
model3.means_ = means
model3.covars_ = covars
```

注意上面有个参数covariance_type，取值为“full”意味所有的 μ, Σ 都需要指定。取值为“spherical”则 Σ 的非对角线元素为0，对角线元素相同。取值为“diag”则 Σ 的非对角线元素为0，对角线元素可以不同，“tied”指所有的隐藏状态对应的观测状态分布使用相同的协方差矩阵 Σ

我们现在跑一跑HMM问题一解码的过程，由于观测状态是二维的，我们用的三维观测序列，所以这里的输入是一个 $3 \times 2 \times 2$ 的张量，代码如下：

```
seen = np.array([[1.1, 2.0], [-1, 2.0], [3, 7]])
logprob, state = model.decode(seen, algorithm="viterbi")
print state
```

输出结果如下：

[0 0 1]

再看看HMM问题一对数概率的计算：

```
print model3.score(seen)
```

输出如下：

-41.1211281377

以上就是用hmmlearn学习HMM的过程。希望可以帮到大家。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

条件随机场CRF(一)从随机场到线性链条件随机场

[条件随机场CRF\(一\)从随机场到线性链条件随机场](#)

[条件随机场CRF\(二\) 前向后向算法评估标记序列概率](#)

[条件随机场CRF\(三\) 模型学习与维特比算法解码](#)

条件随机场(Conditional Random Fields, 以下简称CRF)是给定一组输入序列条件下另一组输出序列的条件概率分布模型，在自然语言处理中得到了广泛应用。本系列主要关注于CRF的特殊形式：线性链(Linear chain) CRF。本文关注与CRF的模型基础。

1.什么样的问题需要CRF模型

和HMM类似，在讨论CRF之前，我们来看看什么样的问题需要CRF模型。这里举一个简单的例子：

假设我们有Bob一天从早到晚的一系列照片，Bob想考考我们，要我们猜这一系列的每张照片对应的活动，比如：工作的照片，吃饭的照片，唱歌的照片等等。一个比较直观的办法就是，我们找到Bob之前的日常生活的一系列照片，然后找Bob问清楚这些照片代表的活动标记，这样我们就可以用监督学习的方法来训练一个分类模型，比如逻辑回归，接着用模型去预测这一天的每张照片最可能的活动标记。

这种办法虽然是可行的，但是却忽略了一个重要的问题，就是这些照片之间的顺序其实是有很大的时间顺序关系的，而用上面的方法则会忽略这种关系。比如我们现在看到了一张Bob闭着嘴的照片，那么这张照片我们怎么标记Bob的活动呢？比较难去打标记。但是如果我们在这一张照片前一点点时间的照片的话，那么这张照片就好标记了。如果在时间序列上前一张的照片里Bob在吃饭，那么这张闭嘴的照片很有可能是在吃饭咀嚼。而如果在时间序列上前一张的照片里Bob在唱歌，那么这张闭嘴的照片很有可能是在唱歌。

为了让我们的分类器表现的更好，可以在标记数据的时候，可以考虑相邻数据的标记信息。这一点，是普通的分类器难以做到的。而这一块，也是CRF比较擅长的地方。

在实际应用中，自然语言处理中的词性标注(POS Tagging)就是非常适合CRF使用的地方。词性标注的目标是给出一个句子中每个词的词性（名词，动词，形容词等）。而这些词的词性往往和上下文的词的词性有关，因此，使用CRF来处理是很适合的，当然CRF不是唯一的选择，也有很多其他的词性标注方法。

2. 从随机场到马尔科夫随机场

首先，我们来看看什么是随机场。“随机场”的名字取的很玄乎，其实理解起来不难。随机场是由若干个位置组成的整体，当给每一个位置中按照某种分布随机赋予一个值之后，其全体就叫做随机场。还是举词性标注的例子：假如我们有一个十个词形成的句子需要做词性标注。这十个词每个词的词性可以在我们已知的词性集合（名词，动词...）中去选择。当我们为每个词选择完词性后，这就形成了一个随机场。

了解了随机场，我们再来看看马尔科夫随机场。马尔科夫随机场是随机场的特例，它假设随机场中某一个位置的赋值仅仅与和它相邻的位置的赋值有关，和与其不相邻的位置的赋值无关。继续举十个词的句子词性标注的例子：如果我们假设所有词的词性只和它相邻的词的词性有关时，这个随机场就特化成了一个马尔科夫随机场。比如第三个词的词性除了与自己本身的位置有关外，只与第二个词和第四个词的词性有关。

3. 从马尔科夫随机场到条件随机场

理解了马尔科夫随机场，再理解CRF就容易了。CRF是马尔科夫随机场的特例，它假设马尔科夫随机场中只有 X 和 Y 两种变量， X 一般是给定的，而 Y 一般是在给定 X 的条件下我们的输出。这样马尔科夫随机场就特化成了条件随机场。在我们十个词的句子词性标注的例子中， X 是词， Y 是词性。因此，如果我们假设它是一个马尔科夫随机场，那么它也就是一个CRF。

对于CRF，我们给出准确的数学语言描述：

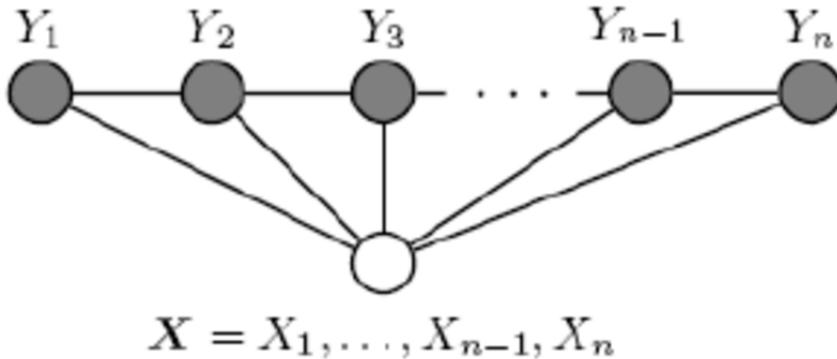
设 X 与 Y 是随机变量， $P(Y|X)$ 是给定 X 时 Y 的条件概率分布，若随机变量 Y 构成的是一个马尔科夫随机场，则称条件概率分布 $P(Y|X)$ 是条件随机场。

4. 从条件随机场到线性链条件随机场

注意在CRF的定义中，我们并没有要求 X 和 Y 有相同的结构。而实现中，我们一般都假设 X 和 Y 有相同的结构，即：

$$X = (X_1, X_2, \dots, X_n), Y = (Y_1, Y_2, \dots, Y_n)$$

我们一般考虑如下图所示的结构： X 和 Y 有相同的结构的CRF就构成了线性链条件随机场(Linear chain Conditional Random Fields,以下简称 linear-CRF)。



在我们的十个词的句子的词性标记中，词有十个，词性也是十个，因此，如果我们假设它是一个马尔科夫随机场，那么它也就是一个linear-CRF。

我们再来看看 linear-CRF的数学定义：

设 $X = (X_1, X_2, \dots, X_n)$, $Y = (Y_1, Y_2, \dots, Y_n)$ 均为线性链表示的随机变量序列，在给定随机变量序列 X 的情况下，随机变量 Y 的条件概率分布 $P(Y|X)$ 构成条件随机场，即满足马尔科夫性：

$$P(Y_i|X, Y_1, Y_2, \dots, Y_n) = P(Y_i|X, Y_{i-1}, Y_{i+1})$$

则称 $P(Y|X)$ 为线性链条件随机场。

5. 线性链条件随机场的参数化形式

对于上一节讲到的linear-CRF，我们如何将其转化为可以学习的机器学习模型呢？这是通过特征函数和其权重系数来定义的。什么是特征函数呢？

在linear-CRF中，特征函数分为两类，第一类是定义在 Y 节点上的节点特征函数，这类特征函数只和当前节点有关，记为：

$$s_l(y_i, x, i), \quad l = 1, 2, \dots, L$$

其中 L 是定义在该节点的节点特征函数的总个数， i 是当前节点在序列的位置。

第二类是定义在 Y 上下文的局部特征函数，这类特征函数只和当前节点和上一个节点有关，记为：

$$t_k(y_{i-1}, y_i, x, i), \quad k = 1, 2, \dots, K$$

其中 K 是定义在该节点的局部特征函数的总个数， i 是当前节点在序列的位置。之所以只有上下文相关的局部特征函数，没有不相邻节点之间的特征函数，是因为我们的linear-CRF满足马尔科夫性。

无论是节点特征函数还是局部特征函数，它们的取值只能是0或者1。即满足特征条件或者不满足特征条件。同时，我们可以为每个特征函数赋予一个权值，用以表达我们对这个特征函数的信任度。假设 t_k 的权重系数是 λ_k , s_l 的权重系数是 μ_l , 则linear-CRF由我们所有的 $t_k, \lambda_k, s_l, \mu_l$ 共同决定。

此时我们得到了linear-CRF的参数化形式如下：

$$P(y|x) = \frac{1}{Z(x)} \exp \left(\sum_{i,k} \lambda_k t_k(y_{i-1}, y_i, x, i) + \sum_{i,l} \mu_l s_l(y_i, x, i) \right)$$

其中， $Z(x)$ 为规范化因子：

$$Z(x) = \sum_y \exp \left(\sum_{i,k} \lambda_k t_k(y_{i-1}, y_i, x, i) + \sum_{i,l} \mu_l s_l(y_i, x, i) \right)$$

回到特征函数本身，每个特征函数定义了一个linear-CRF的规则，则其系数定义了这个规则的可信度。所有的规则和其可信度一起构成了我们的linear-CRF的最终的条件概率分布。

6. 线性链条件随机场实例

这里我们给出一个linear-CRF用于词性标注的实例，为了方便，我们简化了词性的种类。假设输入的都是三个词的句子，即 $X = (X_1, X_2, X_3)$, 输出的词性标记为 $Y = (Y_1, Y_2, Y_3)$, 其中 $Y \in \{1(\text{名词}), 2(\text{动词})\}$

这里只标记出取值为1的特征函数如下：

$$t_1 = t_1(y_{i-1} = 1, y_i = 2, x, i), \quad i = 2, 3, \quad \lambda_1 = 1$$

$$t_2 = t_2(y_1 = 1, y_2 = 1, x, 2) \quad \lambda_2 = 0.5$$

$$t_3 = t_3(y_2 = 2, y_3 = 1, x, 3) \quad \lambda_3 = 1$$

$$t_4 = t_4(y_1 = 2, y_2 = 1, x, 2) \quad \lambda_4 = 1$$

$$t_5 = t_5(y_2 = 2, y_3 = 2, x, 3) \quad \lambda_5 = 0.2$$

$$s_1 = s_1(y_1 = 1, x, 1) \quad \mu_1 = 1$$

$$s_2 = s_2(y_i = 2, x, i), i = 1, 2, \quad \mu_2 = 0.5$$

$$s_3 = s_3(y_i = 1, x, i), i = 2, 3, \quad \mu_3 = 0.8$$

$$s_4 = s_4(y_3 = 2, x, 3) \quad \mu_4 = 0.5$$

求标记(1,2,2)的非规范化概率。

利用linear-CRF的参数化公式，我们有：

$$P(y|x) \propto \exp \left[\sum_{k=1}^5 \lambda_k \sum_{i=2}^3 t_k(y_{i-1}, y_i, x, i) + \sum_{l=1}^4 \mu_l \sum_{i=1}^3 s_l(y_i, x, i) \right]$$

带入(1,2,2)我们有：

$$P(y_1 = 1, y_2 = 2, y_3 = 2 | x) \propto \exp(3.2)$$

7. 线性链条件随机场的简化形式

在上几节里面，我们用 s_l 表示节点特征函数，用 t_k 表示局部特征函数，同时也用了不同的符号表示权重系数，导致表示起来比较麻烦。其实我们可以对特征函数稍加整理，将其统一起来。

假设我们在某一节点我们有 K_1 个局部特征函数和 K_2 个节点特征函数，总共有 $K = K_1 + K_2$ 个特征函数。我们用一个特征函数 $f_k(y_{i-1}, y_i, x, i)$ 来统一表示如下：

$$f_k(y_{i-1}, y_i, x, i) = \begin{cases} t_k(y_{i-1}, y_i, x, i) & k = 1, 2, \dots, K_1 \\ s_l(y_i, x, i) & k = K_1 + l, l = 1, 2, \dots, K_2 \end{cases}$$

对 $f_k(y_{i-1}, y_i, x, i)$ 在各个序列位置求和得到：

$$f_k(y, x) = \sum_{i=1}^n f_k(y_{i-1}, y_i, x, i)$$

同时我们也统一 $f_k(y_{i-1}, y_i, x, i)$ 对应的权重系数 w_k 如下：

$$w_k = \begin{cases} \lambda_k & k = 1, 2, \dots, K_1 \\ \mu_l & k = K_1 + l, l = 1, 2, \dots, K_2 \end{cases}$$

这样，我们的linear-CRF的参数化形式简化为：

$$P(y|x) = \frac{1}{Z(x)} \exp \sum_{k=1}^K w_k f_k(y, x)$$

其中， $Z(x)$ 为规范化因子：

$$Z(x) = \sum_y \exp \sum_{k=1}^K w_k f_k(y, x)$$

如果将上两式中的 w_k 与 f_k 的用向量表示，即：

$$w = (w_1, w_2, \dots, w_K)^T \quad F(y, x) = (f_1(y, x), f_2(y, x), \dots, f_K(y, x))^T$$

则linear-CRF的参数化形式简化为内积形式如下：

$$P_w(y|x) = \frac{\exp(w \bullet F(y, x))}{Z_w(x)} = \frac{\exp(w \bullet F(y, x))}{\sum_y \exp(w \bullet F(y, x))}$$

8. 线性链条件随机场的矩阵形式

将上一节统一后的linear-CRF公式加以整理，我们还可以将linear-CRF的参数化形式写成矩阵形式。为此我们定义一个 $m \times m$ 的矩阵 M ， m 为 y 所有可能的状态的取值个数。 M 定义如下：

$$M_i(x) = [M_i(y_{i-1}, y_i | x)] = [\exp(W_i(y_{i-1}, y_i | x))] = [\exp(\sum_{k=1}^K w_k f_k(y_{i-1}, y_i, x, i))]$$

我们引入起点和终点标记 $y_0 = start$, $y_{n+1} = stop$, 这样，标记序列 y 的非规范化概率可以通过 $n + 1$ 个矩阵元素的乘积得到，即：

$$P_w(y|x) = \frac{1}{Z_w(x)} \prod_{i=1}^{n+1} M_i(y_{i-1}, y_i | x)$$

其中 $Z_w(x)$ 为规范化因子。

以上就是linear-CRF的模型基础，后面我们会讨论linear-CRF和HMM类似的三个问题的求解方法。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

条件随机场CRF(二) 前向后向算法评估标记序列概率

[条件随机场CRF\(一\)从随机场到线性链条件随机场](#)

[条件随机场CRF\(二\) 前向后向算法评估标记序列概率](#)

[条件随机场CRF\(三\) 模型学习与维特比算法解码](#)

在条件随机场CRF(一)中我们总结了CRF的模型，主要是linear-CRF的模型原理。本文就继续讨论linear-CRF需要解决的三个问题：评估，学习和解码。这三个问题和HMM是非常类似的，本文关注于第一个问题：评估。第二个和第三个问题会在下一篇总结。

1. linear-CRF的三个基本问题

在隐马尔科夫模型HMM中，我们讲到了HMM的三个基本问题，而linear-CRF也有三个类似的基本问题。不过和HMM不同，在linear-CRF中，我们对于给出的观测序列 x 是作为一个整体看待的，也就是不会拆开看 (x_1, x_2, \dots) ，因此linear-CRF的问题模型要比HMM简单一些，如果你很熟悉HMM，那么CRF的这三个问题的求解就不难了。

linear-CRF第一个问题是评估，即给定 linear-CRF的条件概率分布 $P(y|x)$ ，在给定输入序列 x 和输出序列 y 时，计算条件概率 $P(y_i|x)$ 和 $P(y_{i-1}, y_i|x)$ 以及对应的期望。本文接下来会详细讨论问题一。

linear-CRF第二个问题是学习，即给定训练数据集 X 和 Y ，学习linear-CRF的模型参数 w_k 和条件概率 $P_w(y|x)$ ，这个问题的求解比HMM的学习算法简单的多，普通的梯度下降法，拟牛顿法都可以解决。

linear-CRF第三个问题是解码，即给定 linear-CRF的条件概率分布 $P(y|x)$ 和输入序列 x ，计算使条件概率最大的输出序列 y 。类似于HMM，使用维特比算法可以很方便的解决这个问题。

2.linear-CRF的前向后向概率概述

要计算条件概率 $P(y_i|x)$ 和 $P(y_{i-1}, y_i|x)$ ，我们也可以使用和HMM类似的方法，使用前向后向算法来完成。首先我们来看前向概率的计算。

我们定义 $\alpha_i(y_i|x)$ 表示序列位置 i 的标记是 y_i 时，在位置 i 之前的部分标记序列的非规范化概率。之所以是非规范化概率是因为我们不想加入一个不影响结果计算的规范化因子 $Z(x)$ 在分母里面。

[在条件随机场CRF\(一\)第八节中](#)，我们定义了下式：

$$M_i(y_{i-1}, y_i|x) = \exp\left(\sum_{k=1}^K w_k f_k(y_{i-1}, y_i, x, i)\right)$$

这个式子定义了在给定 y_{i-1} 时，从 y_{i-1} 转移到 y_i 的非规范化概率。

这样，我们很容易得到序列位置 $i+1$ 的标记是 y_{i+1} 时，在位置 $i+1$ 之前的部分标记序列的非规范化概率 $\alpha_{i+1}(y_{i+1}|x)$ 的递推公式：

$$\alpha_{i+1}(y_{i+1}|x) = \alpha_i(y_i|x) M_{i+1}(y_{i+1}, y_i|x)$$

在起点处，我们定义：

$$\alpha_0(y_0|x) = \begin{cases} 1 & y_0 = start \\ 0 & else \end{cases}$$

假设我们可能的标记总数是 m ，则 y_i 的取值就有 m 个，我们用 $\alpha_i(x)$ 表示这 m 个值组成的前向向量如下：

$$\alpha_i(x) = (\alpha_i(y_i=1|x), \alpha_i(y_i=2|x), \dots, \alpha_i(y_i=m|x))^T$$

同时用矩阵 $M_i(x)$ 表示由 $M_i(y_{i-1}, y_i|x)$ 形成的 $m \times m$ 阶矩阵：

$$M_i(x) = [M_i(y_{i-1}, y_i|x)]$$

这样递推公式可以用矩阵乘积表示：

$$\alpha_{i+1}^T(x) = \alpha_i^T(x) M_i(x)$$

同样的。我们定义 $\beta_i(y_i|x)$ 表示序列位置 i 的标记是 y_i 时，在位置 i 之后的从 $i+1$ 到 n 的部分标记序列的非规范化概率。

这样，我们很容易得到序列位置 $i+1$ 的标记是 y_{i+1} 时，在位置 i 之后的部分标记序列的非规范化概率 $\beta_i(y_i|x)$ 的递推公式：

$$\beta_i(y_i|x) = M_i(y_i, y_{i+1}|x) \beta_{i+1}(y_{i+1}|x)$$

在终点处，我们定义：

$$\beta_{n+1}(y_{n+1}|x) = \begin{cases} 1 & y_{n+1} = stop \\ 0 & else \end{cases}$$

如果用向量表示，则有：

$$\beta_i(x) = M_i(x) \beta_{i+1}(x)$$

由于规范化因子 $Z(x)$ 的表达式是：

$$Z(x) = \sum_{c=1}^m \alpha_n(y_c|x) = \sum_{c=1}^m \beta_1(y_c|x)$$

也可以用向量来表示 $Z(x)$ ：

$$Z(x) = \alpha_n^T(x) \bullet \mathbf{1} = \mathbf{1}^T \bullet \beta_1(x)$$

其中， $\mathbf{1}$ 是 m 维全1向量。

3. linear-CRF的前向后向概率计算

有了前向后向概率的定义和计算方法，我们就很容易计算序列位置 i 的标记是 y_i 时的条件概率 $P(y_i|x)$ ：

$$P(y_i|x) = \frac{\alpha_i^T(y_i|x) \beta_i(y_i|x)}{Z(x)} = \frac{\alpha_i^T(y_i|x) \beta_i(y_i|x)}{\alpha_n^T(x) \bullet \mathbf{1}}$$

也容易计算序列位置 i 的标记是 y_i ，位置 $i-1$ 的标记是 y_{i-1} 时的条件概率 $P(y_{i-1}, y_i|x)$ ：

$$P(y_{i-1}, y_i|x) = \frac{\alpha_{i-1}^T(y_{i-1}|x) M_i(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)} = \frac{\alpha_{i-1}^T(y_{i-1}|x) M_i(y_{i-1}, y_i|x) \beta_i(y_i|x)}{\alpha_n^T(x) \bullet \mathbf{1}}$$

4. linear-CRF的期望计算

有了上一节计算的条件概率，我们也可以很方便的计算联合分布 $P(x, y)$ 与条件分布 $P(y|x)$ 的期望。

特征函数 $f_k(x, y)$ 关于条件分布 $P(y|x)$ 的期望表达式是：

$$E_{P(y|x)}[f_k] = E_{P(y|x)}[f_k(y, x)] \quad (1)$$

$$= \sum_{i=1}^{n+1} \sum_{y_{i-1}} \sum_{y_i} P(y_{i-1}, y_i|x) f_k(y_{i-1}, y_i, x, i) \quad (2)$$

$$= \sum_{i=1}^{n+1} \sum_{y_{i-1}} \sum_{y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}^T(y_{i-1}|x) M_i(y_{i-1}, y_i|x) \beta_i(y_i|x)}{\alpha_n^T(x) \bullet \mathbf{1}} \quad (3)$$

同样可以计算联合分布 $P(x, y)$ 的期望：

$$E_{P(x,y)}[f_k] = \sum_{x,y} P(x, y) \sum_{i=1}^{n+1} f_k(y_{i-1}, y_i, x, i) \quad (4)$$

$$= \sum_x \bar{P}(x) \sum_y P(y|x) \sum_{i=1}^{n+1} f_k(y_{i-1}, y_i, x, i) \quad (5)$$

$$= \sum_x \bar{P}(x) \sum_{i=1}^{n+1} \sum_{y_{i-1}} \sum_{y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}^T(y_{i-1}|x) M_i(y_{i-1}, y_i|x) \beta_i(y_i|x)}{\alpha_n^T(x) \bullet \mathbf{1}} \quad (6)$$

假设一共有 K 个特征函数，则 $k = 1, 2, \dots, K$

5. linear-CRF前向后向算法总结

以上就是linear-CRF的前向后向算法，个人觉得比HMM简单的多，因此大家如果理解了HMM的前向后向算法，这一篇是很容易理解的。

注意到我们上面的非规范化概率 $M_{i+1}(y_{i+1}, y_i|x)$ 起的作用和HMM中的隐藏状态转移概率很像。但是这儿的概率是非规范化的，也就是不强制要求所有的状态的概率和为1。而HMM中的隐藏状态转移概率也规范化的。从这一点看，linear-CRF对序列状态转移的处理要比HMM灵活。

条件随机场CRF(三) 模型学习与维特比算法解码

[条件随机场CRF\(一\)从随机场到线性链条件随机场](#)

[条件随机场CRF\(二\) 前向后向算法评估标记序列概率](#)

条件随机场CRF(三) 模型学习与维特比算法解码

在CRF系列的前两篇，我们总结了CRF的模型基础与第一个问题的求解方法，本文我们关注于linear-CRF的第二个问题与第三个问题的求解。第二个问题是模型参数学习的问题，第三个问题是维特比算法解码的问题。

1. linear-CRF模型参数学习思路

在linear-CRF模型参数学习问题中，我们给定训练数据集 X 和对应的标记序列 Y ， K 个特征函数 $f_k(x, y)$ ，需要学习linear-CRF的模型参数 w_k 和条件概率 $P_w(y|x)$ ，其中条件概率 $P_w(y|x)$ 和模型参数 w_k 满足一下关系：

$$P_w(y|x) = P(y|x) = \frac{1}{Z_w(x)} \exp \sum_{k=1}^K w_k f_k(x, y) = \frac{\exp \sum_{k=1}^K w_k f_k(x, y)}{\sum_y \exp \sum_{k=1}^K w_k f_k(x, y)}$$

所以我们的目标就是求出所有的模型参数 w_k ，这样条件概率 $P_w(y|x)$ 可以从上式计算出来。

求解这个问题有很多思路，比如梯度下降法，牛顿法，拟牛顿法。同时，这个模型中 $P_w(y|x)$ 的表达式和最大熵模型原理小结中的模型一样，也可以使用最大熵模型中使用的改进的迭代尺度法(improved iterative scaling, IIS)来求解。

下面我们只简要介绍用梯度下降法的求解思路。

2. linear-CRF模型参数学习之梯度下降法求解

在使用梯度下降法求解模型参数之前，我们需要定义我们的优化函数，一般极大化条件分布 $P_w(y|x)$ 的对数似然函数如下：

$$L(w) = \log \prod_{x,y} P_w(y|x)^{\bar{P}(x,y)} = \sum_{x,y} \bar{P}(x,y) \log P_w(y|x)$$

其中 $\bar{P}(x,y)$ 为经验分布，可以从先验知识和训练集样本中得到，这点和最大熵模型类似。为了使用梯度下降法，我们现在极小化 $f(w) = -L(P_w)$ 如下：

$$f(w) = - \sum_{x,y} \bar{P}(x,y) \log P_w(y|x) \quad (1)$$

$$= \sum_{x,y} \bar{P}(x,y) \log Z_w(x) - \sum_{x,y} \bar{P}(x,y) \sum_{k=1}^K w_k f_k(x,y) \quad (2)$$

$$= \sum_x \bar{P}(x) \log Z_w(x) - \sum_{x,y} \bar{P}(x,y) \sum_{k=1}^K w_k f_k(x,y) \quad (3)$$

$$= \sum_x \bar{P}(x) \log \sum_y \exp \sum_{k=1}^K w_k f_k(x,y) - \sum_{x,y} \bar{P}(x,y) \sum_{k=1}^K w_k f_k(x,y) \quad (4)$$

对 w 求导可以得到：

$$\frac{\partial f(w)}{\partial w} = \sum_{x,y} \bar{P}(x) P_w(y|x) f(x,y) - \sum_{x,y} \bar{P}(x,y) f(x,y)$$

有了 w 的导数表达书，就可以用梯度下降法来迭代求解最优的 w 了。注意在迭代过程中，每次更新 w 后，需要同步更新 $P_w(x,y)$ ，以用于下一次迭代的梯度计算。

梯度下降法的过程这里就不累述了，如果不熟悉梯度下降算法过程建议阅读之前写的梯度下降 ([Gradient Descent](#)) 小结。以上就是linear-CRF模型参数学习之梯度下降法求解思路总结。

3. linear-CRF模型维特比算法解码思路

现在我们来看linear-CRF的第三个问题：解码。在这个问题中，给定条件随机场的条件概率 $P(y|x)$ 和一个观测序列 x ,要求出满足 $P(y|x)$ 最大的序列 y 。

这个解码算法最常用的还是和HMM解码类似的维特比算法。到目前为止，我已经在三个地方讲到了维特比算法，第一个是文本挖掘的分词原理中用于中文分词，第二个是隐马尔科夫模型HMM（四）维特比算法解码隐藏状态序列中用于HMM解码。第三个就是这一篇了。

维特比算法本身是一个动态规划算法，利用了两个局部状态和对应的递推公式，从局部递推到整体，进而得解。对于具体不同的问题，仅仅是这两个局部状态的定义和对应的递推公式不同而已。由于在之前已详述维特比算法，这里就是做一个简略的流程描述。

对于我们linear-CRF中的维特比算法，我们的第一个局部状态定义为 $\delta_i(l)$,表示在位置*i*标记*l*各个可能取值(1,2...m)对应的非规范化概率的最大值。之所以用非规范化概率是，规范化因子 $Z(x)$ 不影响最大值的比较。根据 $\delta_i(l)$ 的定义，我们递推在位置*i+1*标记*l*的表达式为：

$$\delta_{i+1}(l) = \max_{1 \leq j \leq m} \{ \delta_i(j) + \sum_{k=1}^K w_k f_k(y_i = j, y_{i+1} = l, x, i) \}, l = 1, 2, \dots, m$$

和HMM的维特比算法类似，我们需要用另一个局部状态 $\Psi_{i+1}(l)$ 来记录使 $\delta_{i+1}(l)$ 达到最大的位置*i*的标记取值,这个值用来最终回溯最优解， $\Psi_{i+1}(l)$ 的递推表达式为：

$$\Psi_{i+1}(l) = \arg \max_{1 \leq j \leq m} \{ \delta_i(j) + \sum_{k=1}^K w_k f_k(y_i = j, y_{i+1} = l, x, i) \}, l = 1, 2, \dots, m$$

4. linear-CRF模型维特比算法流程

现在我们总结下 linear-CRF模型维特比算法流程：

输入：模型的 K 个特征函数，和对应的 K 个权重。观测序列 $x = (x_1, x_2, \dots, x_n)$,可能的标记个数 m

输出：最优标记序列 $y^* = (y_1^*, y_2^*, \dots, y_n^*)$

1) 初始化：

$$\delta_1(l) = \sum_{k=1}^K w_k f_k(y_0 = start, y_1 = l, x, i) \}, l = 1, 2, \dots, m$$

$$\Psi_1(l) = start, l = 1, 2, \dots, m$$

2) 对于 $i = 1, 2, \dots, n-1$,进行递推：

$$\delta_{i+1}(l) = \max_{1 \leq j \leq m} \{ \delta_i(j) + \sum_{k=1}^K w_k f_k(y_i = j, y_{i+1} = l, x, i) \}, l = 1, 2, \dots, m$$

$$\Psi_{i+1}(l) = \arg \max_{1 \leq j \leq m} \{ \delta_i(j) + \sum_{k=1}^K w_k f_k(y_i = j, y_{i+1} = l, x, i) \}, l = 1, 2, \dots, m$$

3) 终止：

$$y_n^* = \arg \max_{1 \leq j \leq m} \delta_n(j)$$

4)回溯：

$$y_i^* = \Psi_{i+1}(y_{i+1}^*), i = n-1, n-2, \dots, 1$$

最终得到最优标记序列 $y^* = (y_1^*, y_2^*, \dots, y_n^*)$

5. linear-CRF模型维特比算法实例

下面用一个具体的例子来描述 linear-CRF模型维特比算法，例子的模型和CRF系列第一篇中一样，都来源于《统计学习方法》。

假设输入的都是三个词的句子，即 $X = (X_1, X_2, X_3)$,输出的词性标记为 $Y = (Y_1, Y_2, Y_3)$,其中 $Y \in \{1(\text{名词}), 2(\text{动词})\}$

这里只标记出取值为1的特征函数如下：

$$t_1 = t_1(y_{i-1} = 1, y_i = 2, x, i), i = 2, 3, \lambda_1 = 1$$

$$t_2 = t_2(y_1 = 1, y_2 = 1, x, 2) \lambda_2 = 0.5$$

$$t_3 = t_3(y_2 = 2, y_3 = 1, x, 3) \lambda_3 = 1$$

$$t_4 = t_4(y_1 = 2, y_2 = 1, x, 2) \lambda_4 = 1$$

$$t_5 = t_5(y_2 = 2, y_3 = 2, x, 3) \lambda_5 = 0.2$$

$$s_1 = s_1(y_1 = 1, x, 1) \quad \mu_1 = 1$$

$$s_2 = s_2(y_i = 2, x, i), i = 1, 2, \quad \mu_2 = 0.5$$

$$s_3 = s_3(y_i = 1, x, i), i = 2, 3, \quad \mu_3 = 0.8$$

$$s_4 = s_4(y_3 = 2, x, 3) \quad \mu_4 = 0.5$$

求标记(1,2,2)的最可能的标记序列。

首先初始化：

$$\delta_1(1) = \mu_1 s_1 = 1 \quad \delta_1(2) = \mu_2 s_2 = 0.5 \quad \Psi_1(1) = \Psi_1(2) = start$$

接下来开始递推，先看位置2的：

$$\delta_2(1) = \max\{\delta_1(1) + t_2 \lambda_2 + \mu_3 s_3, \delta_1(2) + t_4 \lambda_4\} = \max\{1 + 0.5 + 0.8, 0.5 + 1\} = 2.3 \quad \Psi_2(1) = 1$$

$$\delta_2(2) = \max\{\delta_1(1) + t_1 \lambda_1 + \mu_2 s_2, \delta_1(2) + \mu_2 s_2\} = \max\{1 + 0.5 + 0.8, 0.5 + 0.5\} = 2.5 \quad \Psi_2(2) = 1$$

再看位置3的：

$$\delta_3(1) = \max\{\delta_2(1) + \mu_3 s_3, \delta_2(2) + t_3 \lambda_3 + \mu_3 s_3\} = \max\{2.3 + 0.8, 2.5 + 1 + 0.8\} = 4.3$$

$$\Psi_3(1) = 2$$

$$\delta_3(2) = \max\{\delta_2(1) + t_1 \lambda_1 + \mu_4 s_4, \delta_2(2) + t_5 \lambda_5 + \mu_4 s_4\} = \max\{2.3 + 1 + 0.5, 2.5 + 0.2 + 0.5\} = 3.8$$

$$\Psi_3(2) = 1$$

最终得到 $y_3^* = \arg \max\{\delta_3(1), \delta_3(2)\}$, 递推回去, 得到：

$$y_2^* = \Psi_3(1) = 2 \quad y_1^* = \Psi_2(2) = 1$$

即最终的结果为(1, 2, 1), 即标记为(名词, 动词, 名词)。

6.linear-CRF vs HMM

linear-CRF模型和HMM模型有很多相似之处, 尤其是其三个典型问题非常类似, 除了模型参数学习的问题求解方法不同以外, 概率估计问题和解码问题使用的算法思想基本也是相同的。同时, 两者都可以用于序列模型, 因此都广泛用于自然语言处理的各个方面。

现在来看看两者的不同点。最大的不同点是linear-CRF模型是判别模型, 而HMM是生成模型, 即linear-CRF模型要优化求解的是条件概率 $P(y|x)$, 则HMM要求解的是联合分布 $P(x, y)$ 。第二, linear-CRF是利用最大熵模型的思路去建立条件概率模型, 对于观测序列并没有做马尔科夫假设。而HMM是在对观测序列做了马尔科夫假设的前提下建立联合分布的模型。

最后想说的是, 只有linear-CRF模型和HMM模型才是可以比较讨论的。但是linear-CRF是CRF的一个特例, CRF本身是一个可以适用于很复杂条件概率的模型, 因此理论上CRF的使用范围要比HMM广泛的多。

以上就是CRF系列的所有内容。

(欢迎转载, 转载请注明出处。欢迎沟通交流 : pinard.liu@ericsson.com)

word2vec原理(一) CBOW与Skip-Gram模型基础

[word2vec原理\(一\) CBOW与Skip-Gram模型基础](#)

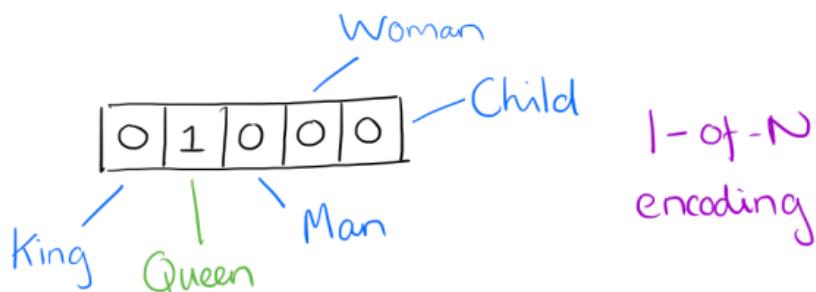
[word2vec原理\(二\) 基于Hierarchical Softmax的模型](#)

[word2vec原理\(三\) 基于Negative Sampling的模型](#)

word2vec是google在2013年推出的一个NLP工具，它的特点是将所有的词向量化，这样词与词之间就可以定量的去度量他们之间的关系，挖掘词之间的联系。虽然源码是开源的，但是谷歌的代码库国内无法访问，因此本文的讲解word2vec原理以Github上的word2vec代码为准。本文关注于word2vec的基础知识。

1. 词向量基础

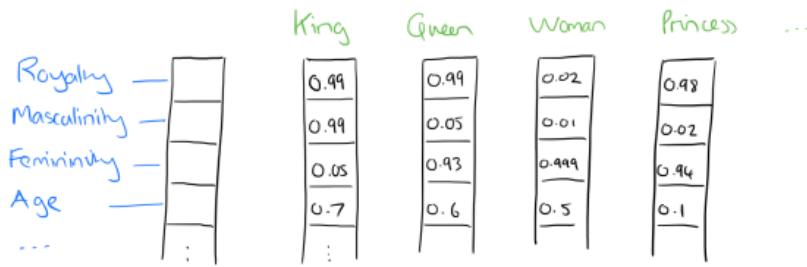
用词向量来表示词并不是word2vec的首创，在很久之前就出现了。最早的词向量是很冗长的，它使用的是词向量维度大小为整个词汇表的大小，对于每个具体的词汇表中的词，将对应的位置置为1。比如我们有下面的5个词组成的词汇表，词"Queen"的序号为2，那么它的词向量就是(0, 1, 0, 0, 0)。同样的道理，词"Woman"的词向量就是(0, 0, 0, 1, 0)。这种词向量的编码方式我们一般叫做1-of-N representation或者one hot representation。



One hot representation用来表示词向量非常简单，但是却有很多问题。最大的问题是我们的词汇表一般都非常大，比如达到百万级别，这样每个词都用百万维的向量来表示简直是内存的灾难。这样的向量其实除了一个位置是1，其余的位置全部都是0，表达的效率不高，能不能把词向量的维度变小呢？

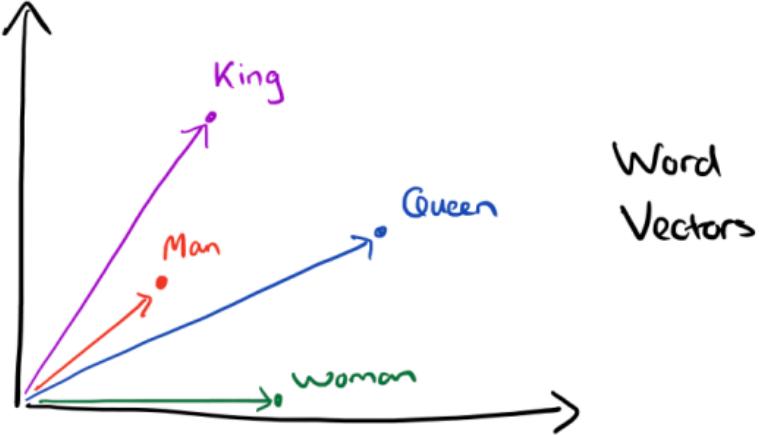
Distributed representation可以解决One hot representation的问题，它的思路是通过训练，将每个词都映射到一个较短的词向量上来。所有的这些词向量就构成了向量空间，进而可以用普通的统计学的方法来研究词与词之间的关系。这个较短的词向量维度是多大呢？这个一般需要我们在训练时自己来指定。

比如下图我们将词汇表里的词用"Royalty", "Masculinity", "Femininity"和"Age"4个维度来表示，King这个词对应的词向量可能是(0.99, 0.99, 0.05, 0.7)。当然在实际情况中，我们并不能对词向量的每个维度做一个很好的解释。



有了用Distributed representation表示的较短的词向量，我们就可以较容易的分析词之间的关系了，比如我们将词的维度降维到2维，有一个有趣的研究表明，用下图的词向量表示我们的词时，我们可以发现：

$$\vec{\text{King}} - \vec{\text{Man}} + \vec{\text{Woman}} = \vec{\text{Queen}}$$



可见我们只要得到了词汇表里所有词对应的词向量，那么我们就可以做很多有趣的事情了。不过，怎么训练得到合适的词向量呢？一个很常见的方法是使用神经网络语言模型。

2. CBOW与Skip-Gram用于神经网络语言模型

在word2vec出现之前，已经有用神经网络DNN来用训练词向量进而处理词与词之间的关系了。采用的方法一般是一个三层的神经网络结构（当然也可以多层），分为输入层，隐藏层和输出层（softmax层）。

这个模型是如何定义数据的输入和输出呢？一般分为CBOW(Continuous Bag-of-Words)与Skip-Gram两种模型。

CBOW模型的训练输入是某一个特征词的上下文相关的词对应的词向量，而输出就是这特定的一个词的词向量。比如下面这段话，我们的上下文大小取值为4，特定的这个词是"Learning"，也就是我们需要的输出词向量，上下文对应的词有8个，前后各4个，这8个词是我们模型的输入。由于CBOW使用的是词袋模型，因此这8个词都是平等的，也就是不考虑他们和我们关注的词之间的距离大小，只要在我们上下文之内即可。



这样我们这个CBOW的例子里，我们的输入是8个词向量，输出是所有词的softmax概率（训练的目标是期望训练样本特定词对应的softmax概率最大），对应的CBOW神经网络模型输入层有8个神经元，输出层有词汇表大小个神经元。隐藏层的神经元个数我们可以自己指定。通过DNN的反向传播算法，我们可以求出DNN模型的参数，同时得到所有的词对应的词向量。这样当我们有新的需求，要求出某8个词对应的最可能的输出中心词时，我们可以通过一次DNN前向传播算法并通过softmax激活函数找到概率最大的词对应的神经元即可。

Skip-Gram模型和CBOW的思路是反着来的，即输入是特定的一个词的词向量，而输出是特定词对应的上下文词向量。还是上面的例子，我们的上下文大小取值为4，特定的这个词"Learning"是我们的输入，而这8个上下文词是我们的输出。

这样我们这个Skip-Gram的例子里，我们的输入是特定词，输出是softmax概率排前8的8个词，对应的Skip-Gram神经网络模型输入层有1个神经元，输出层有词汇表大小个神经元。隐藏层的神经元个数我们可以自己指定。通过DNN的反向传播算法，我们可以求出DNN模型的参数，同时得到所有的词对应的词向量。这样当我们有新的需求，要求出某1个词对应的最可能的8个上下文词时，我们可以通过一次DNN前向传播算法得到概率大小排前8的softmax概率对应的神经元所对应的词即可。

以上就是神经网络语言模型中如何用CBOW与Skip-Gram来训练模型与得到词向量的大概过程。但是这和word2vec中用CBOW与Skip-Gram来训练模型与得到词向量的过程有很多的不同。

word2vec为什么不用现成的DNN模型，要继续优化出新方法呢？最主要的问题是DNN模型的这个处理过程非常耗时。我们的词汇表一般在百万级别以上，这意味着我们DNN的输出层需要进行softmax计算各个词的输出概率的计算量很大。有没有简化一点点的方法呢？

3. word2vec基础之霍夫曼树

word2vec也使用了CBOW与Skip-Gram来训练模型与得到词向量，但是并没有使用传统的DNN模型。最先优化使用的数据结构是用霍夫曼树来代替隐藏层和输出层的神经元，霍夫曼树的叶子节点起到输出层神经元的作用，叶子节点的个数即为词汇表的大小。而内部节点则起到隐藏层神经元的作用。

具体如何用霍夫曼树来进行CBOW和Skip-Gram的训练我们在下一节讲，这里我们先复习下霍夫曼树。

霍夫曼树的建立其实并不难，过程如下：

输入：权值为 (w_1, w_2, \dots, w_n) 的n个节点

输出：对应的霍夫曼树

1) 将 (w_1, w_2, \dots, w_n) 看做是有n棵树的森林，每个树仅有一个节点。

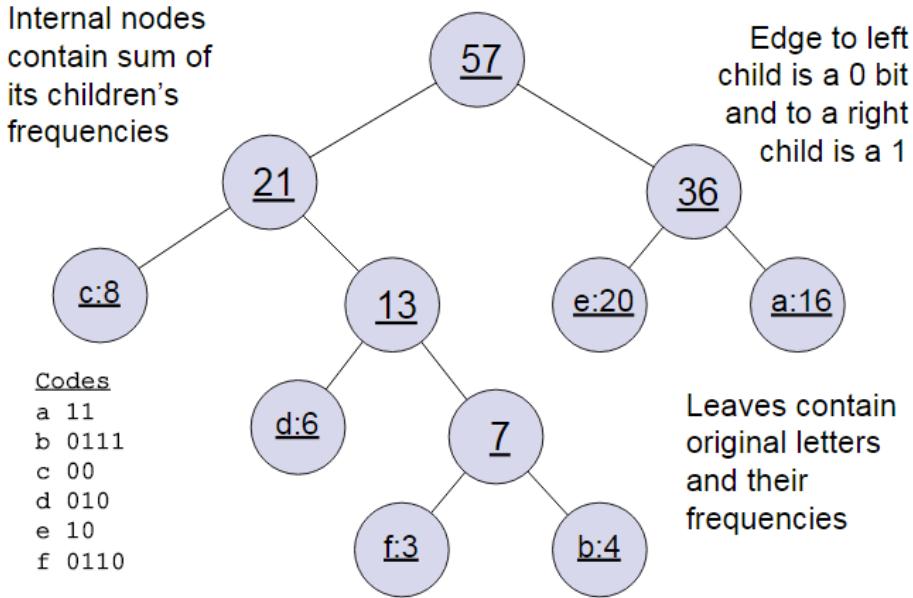
2) 在森林中选择根节点权值最小的两棵树进行合并，得到一个新的树，这两棵树分布作为新树的左右子树。新树的根节点权重为左右子树的根节点权重之和。

3) 将之前的根节点权值最小的两棵树从森林删除，并把新树加入森林。

4) 重复步骤2) 和3) 直到森林里只有一棵树为止。

下面我们用一个具体的例子来说明霍夫曼树建立的过程，我们有(a,b,c,d,e,f)共6个节点，节点的权值分布是(16,4,8,6,20,3)。

首先是b和f合并，得到的新树根节点权重是7.此时森林里5棵树，根节点权重分别是16,8,6,20,7。此时根节点权重最小的6,7合并，得到新子树，依次类推，最终得到下面的霍夫曼树。



那么霍夫曼树有什么好处呢？一般得到霍夫曼树后我们会对叶子节点进行霍夫曼编码，由于权重高的叶子节点越靠近根节点，而权重低的叶子节点会远离根节点，这样我们的高权重节点编码值较短，而低权重值编码值较长。这保证的树的带权路径最短，也符合我们的信息论，即我们希望越常用的词拥有更短的编码。如何编码呢？一般对于一个霍夫曼树的节点（根节点除外），可以约定左子树编码为0，右子树编码为1.如上图，则可以得到c的编码是00。

在word2vec中，约定编码方式和上面的例子相反，即约定左子树编码为1，右子树编码为0，同时约定左子树的权重不小于右子树的权重。

我们在下一节的Hierarchical Softmax中再继续讲使用霍夫曼树和DNN语言模型相比的好处以及如何训练CBOW&Skip-Gram模型。

（欢迎转载，转载请注明出处。欢迎沟通交流：pinard.liu@ericsson.com）

word2vec原理(二) 基于Hierarchical Softmax的模型

[word2vec原理\(一\) CBOW与Skip-Gram模型基础](#)

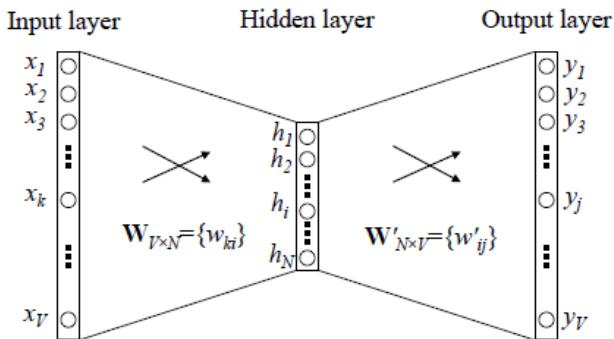
[word2vec原理\(二\) 基于Hierarchical Softmax的模型](#)

[word2vec原理\(三\) 基于Negative Sampling的模型](#)

在[word2vec原理\(一\) CBOW与Skip-Gram模型基础](#)中，我们讲到了使用神经网络的方法来得到词向量语言模型的原理和一些问题，现在我们开始关注word2vec的语言模型如何改进传统的神经网络的方法。由于word2vec有两种改进方法，一种是基于Hierarchical Softmax的，另一种是基于Negative Sampling的。本文关注于基于Hierarchical Softmax的改进方法，在下一篇讨论基于Negative Sampling的改进方法。

1. 基于Hierarchical Softmax的模型概述

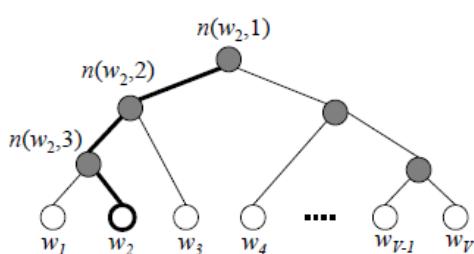
我们先回顾下传统的神经网络词向量语言模型，里面一般有三层，输入层（词向量），隐藏层和输出层（softmax层）。里面最大的问题在于从隐藏层到输出的softmax层的计算量很大，因为要计算所有词的softmax概率，再去找概率最大的值。这个模型如下图所示。其中 V 是词汇表的大小，



word2vec对这个模型做了改进，首先，对于从输入层到隐藏层的映射，没有采取神经网络的线性变换加激活函数的方法，而是采用简单的对所有输入词向量求和并取平均的方法。比如输入的是三个4维词向量： $(1, 2, 3, 4), (9, 6, 11, 8), (5, 10, 7, 12)$ ，那么我们word2vec映射后的词向量就是 $(5, 6, 7, 8)$ 。由于这里是从多个词向量变成了一个词向量。

第二个改进就是从隐藏层到输出的softmax层这里的计算量个改进。为了避免要计算所有词的softmax概率，word2vec采样了霍夫曼树来代替从隐藏层到输出softmax层的映射。我们在上一节已经介绍了霍夫曼树的原理。如何映射呢？这里就是理解word2vec的关键所在了。

由于我们把之前所有都要计算的从输出softmax层的概率计算变成了一颗二叉霍夫曼树，那么我们的softmax概率计算只需要沿着树形结构进行就可以了。如下图所示，我们可以沿着霍夫曼树从根节点一直走到我们的叶子节点的词 w_2 。



和之前的神经网络语言模型相比，我们的霍夫曼树的所有内部节点就类似之前神经网络隐藏层的神经元，其中，根节点的词向量对应我们的投影后的词向量，而所有叶子节点就类似于之前神经网络softmax输出层的神经元，叶子节点的个数就是词汇表的大小。在霍夫曼树中，隐藏层到输出层的softmax映射不是一下子完成的，而是沿着霍夫曼树一步一步完成的，因此这种softmax取名为“Hierarchical Softmax”。

如何“沿着霍夫曼树一步步完成”呢？在word2vec中，我们采用了二元逻辑回归的方法，即规定沿着左子树走，那么就是负类(霍夫曼树编码1)，沿着右子树走，那么就是正类(霍夫曼树编码0)。判别正类和负类的方法是使用sigmoid函数，即：

$$P(+)=\sigma(x_w^T \theta) = \frac{1}{1+e^{-x_w^T \theta}}$$

其中 x_w 是当前内部节点的词向量，而 θ 则是我们需要从训练样本求出的逻辑回归的模型参数。

使用霍夫曼树有什么好处呢？首先，由于是二叉树，之前计算量为 V ，现在变成了 $\log V$ 。第二，由于使用霍夫曼树是高频的词靠近树根，这样高频词需要更少的时间会被找到，这符合我们的贪心优化思想。

容易理解，被划分为左子树而成为负类的概率为 $P(-) = 1 - P(+)$ 。在某一个内部节点，要判断是沿左子树还是右子树走的标准就是看 $P(-), P(+)$ 谁的概率值大。而控制 $P(-), P(+)$ 谁的概率值大的因素一个是当前节点的词向量，另一个是当前节点的模型参数 θ 。

对于上图中的 w_2 ，如果它是一个训练样本的输出，那么我们期望对于里面的隐藏节点 $n(w_2, 1)$ 的 $P(-)$ 概率大， $n(w_2, 2)$ 的 $P(-)$ 概率大， $n(w_2, 3)$ 的 $P(+)$ 概率大。

回到基于Hierarchical Softmax的word2vec本身，我们的目标就是找到合适的所有节点的词向量和所有内部节点 θ ，使训练样本达到最大似然。那么如何达到最大似然呢？

2. 基于Hierarchical Softmax的模型梯度计算

我们使用最大似然法来寻找所有节点的词向量和所有内部节点 θ 。先拿上面的 w_2 例子来看，我们期望最大化下面的似然函数：

$$\prod_{i=1}^3 P(n(w_i), i) = \left(1 - \frac{1}{1+e^{-x_w^T \theta_1}}\right) \left(1 - \frac{1}{1+e^{-x_w^T \theta_2}}\right) \frac{1}{1+e^{-x_w^T \theta_3}}$$

对于所有的训练样本，我们期望最大化所有样本的似然函数乘积。

为了便于我们后面一般化的描述，我们定义输入的词为 w ，其从输入层词向量求和平均后的霍夫曼树根节点词向量为 x_w ，从根节点到 w 所在的叶子节点，包含的节点总数为 l_w ， w 在霍夫曼树中从根节点开始，经过的第 i 个节点表示为 p_i^w ，对应的霍夫曼编码为 $d_i^w \in \{0, 1\}$ ，其中 $i = 2, 3, \dots, l_w$ 。而该节点对应的模型参数表示为 θ_i^w ，其中 $i = 1, 2, \dots, l_w - 1$ ，没有 $i = l_w$ 是因为模型参数仅仅针对于霍夫曼树的内部节点。

定义 w 经过的霍夫曼树某一个节点 j 的逻辑回归概率为 $P(d_j^w | x_w, \theta_{j-1}^w)$ ，其表达式为：

$$P(d_j^w | x_w, \theta_{j-1}^w) = \begin{cases} \sigma(x_w^T \theta_{j-1}^w) & d_j^w = 0 \\ 1 - \sigma(x_w^T \theta_{j-1}^w) & d_j^w = 1 \end{cases}$$

那么对于某一个目标输出词 w ，其最大似然为：

$$\prod_{j=2}^{l_w} P(d_j^w | x_w, \theta_{j-1}^w) = \prod_{j=2}^{l_w} [\sigma(x_w^T \theta_{j-1}^w)]^{1-d_j^w} [1 - \sigma(x_w^T \theta_{j-1}^w)]^{d_j^w}$$

在word2vec中，由于使用的是随机梯度上升法，所以并没有把所有样本的似然乘起来得到真正的训练集最大似然，仅仅每次只用一个样本更新梯度，这样做的目的是减少梯度计算量。这样我们可以得到 w 的对数似然函数 L 如下：

$$L = \log \prod_{j=2}^{l_w} P(d_j^w | x_w, \theta_{j-1}^w) = \sum_{j=2}^{l_w} ((1 - d_j^w) \log[\sigma(x_w^T \theta_{j-1}^w)] + d_j^w \log[1 - \sigma(x_w^T \theta_{j-1}^w)])$$

要得到模型中 w 词向量和内部节点的模型参数 θ ，我们使用梯度上升法即可。首先我们求模型参数 θ_{j-1}^w 的梯度：

$$\frac{\partial L}{\partial \theta_{j-1}^w} = (1 - d_j^w) \frac{(\sigma(x_w^T \theta_{j-1}^w)(1 - \sigma(x_w^T \theta_{j-1}^w))x_w - d_j^w \frac{(\sigma(x_w^T \theta_{j-1}^w)(1 - \sigma(x_w^T \theta_{j-1}^w))}{1 - \sigma(x_w^T \theta_{j-1}^w)}x_w)}{\sigma(x_w^T \theta_{j-1}^w)} \quad (1)$$

$$= (1 - d_j^w)(1 - \sigma(x_w^T \theta_{j-1}^w))x_w - d_j^w \sigma(x_w^T \theta_{j-1}^w)x_w \quad (2)$$

$$= (1 - d_j^w - \sigma(x_w^T \theta_{j-1}^w))x_w \quad (3)$$

如果大家看过之前写的逻辑回归原理小结，会发现这里的梯度推导过程基本类似。

同样的方法，可以求出 x_w 的梯度表达式如下：

$$\frac{\partial L}{\partial x_w} = (1 - d_j^w - \sigma(x_w^T \theta_{j-1}^w))\theta_{j-1}^w$$

有了梯度表达式，我们就可以用梯度上升法进行迭代来一步步的求解我们需要的所有的 θ_{j-1}^w 和 x_w 。

3. 基于Hierarchical Softmax的CBOW模型

由于word2vec有两种模型：CBOW和Skip-Gram，我们先看看基于CBOW模型时，Hierarchical Softmax如何使用。

首先我们要定义词向量的维度大小 M ，以及CBOW的上下文大小 $2c$ ，这样我们对于训练样本中的每一个词，其前面的 c 个词和后面的 c 个词作为CBOW模型的输入，该词本身作为样本的输出，期望softmax概率最大。

在做CBOW模型前，我们需要先将词汇表建立成一颗霍夫曼树。

对于从输入层到隐藏层（投影层），这一步比较简单，就是对 w 周围的 $2c$ 个词向量求和取平均即可，即：

$$x_w = \frac{1}{2c} \sum_{i=1}^{2c} x_i$$

第二步，通过梯度上升法来更新我们的 θ_{j-1}^w 和 x_w ，注意这里的 x_w 是由 $2c$ 个词向量相加而成，我们做梯度更新完毕后会用梯度项直接更新原始的各个 $x_i (i = 1, 2, \dots, 2c)$ ，即：

$$\begin{aligned}\theta_{j-1}^w &= \theta_{j-1}^w + \eta(1 - d_j^w - \sigma(x_w^T \theta_{j-1}^w)) x_w \\ x_w &= x_w + \eta(1 - d_j^w - \sigma(x_w^T \theta_{j-1}^w)) \theta_{j-1}^w (i = 1, 2, \dots, 2c)\end{aligned}$$

其中 η 为梯度上升法的步长。

这里总结下基于Hierarchical Softmax的CBOW模型算法流程，梯度迭代使用了随机梯度上升法：

输入：基于CBOW的语料训练样本，词向量的维度大小 M ，CBOW的上下文大小 $2c$ ，步长 η

输出：霍夫曼树的内部节点模型参数 θ ，所有的词向量 w

1. 基于语料训练样本建立霍夫曼树。
2. 随机初始化所有的模型参数 θ ，所有的词向量 w
3. 进行梯度上升迭代过程，对于训练集中的每一个样本($context(w), w$)做如下处理：

a) $e = 0$ ，计算 $x_w = \frac{1}{2c} \sum_{i=1}^{2c} x_i$

b) for $j = 2$ to l_w ，计算：

$$f = \sigma(x_w^T \theta_{j-1}^w)$$

$$g = (1 - d_j^w - f)\eta$$

$$e = e + g\theta_{j-1}^w$$

$$\theta_{j-1}^w = \theta_{j-1}^w + gx_w$$

c) 对于 $context(w)$ 中的每一个词向量 x_i (共 $2c$ 个)进行更新：

$$x_i = x_i + e$$

d) 如果梯度收敛，则结束梯度迭代，否则回到步骤3继续迭代。

4. 基于Hierarchical Softmax的Skip-Gram模型

现在我们先看看基于Skip-Gram模型时，Hierarchical Softmax如何使用。此时输入的只有一个词 w ，输出的为 $2c$ 个词向量 $context(w)$ 。

我们对于训练样本中的每一个词，该词本身作为样本的输入，其前面的 c 个词和后面的 c 个词作为Skip-Gram模型的输出，期望这些词的softmax概率比其他的词大。

Skip-Gram模型和CBOW模型其实是反过来的，在上一篇已经讲过。

在做CBOW模型前，我们需要先将词汇表建立成一颗霍夫曼树。

对于从输入层到隐藏层（投影层），这一步比CBOW简单，由于只有一个词，所以，即 x_w 就是词 w 对应的词向量。

第二步，通过梯度上升法来更新我们的 θ_{j-1}^w 和 x_w ，注意这里的 x_w 周围有 $2c$ 个词向量，此时如果我们期望 $P(x_i | x_w), i = 1, 2, \dots, 2c$ 最大。此时我们注意到由于上下文是相互的，在期望 $P(x_i | x_w), i = 1, 2, \dots, 2c$ 最大化的同时，反过来我们也期望 $P(x_w | x_i), i = 1, 2, \dots, 2c$ 最大。那么是使用 $P(x_i | x_w)$ 好还是 $P(x_w | x_i)$ 好呢，word2vec使用了后者，这样做的好处就是在一次迭代时，我们不是更新 x_w 一个词，而是 $x_i, i = 1, 2, \dots, 2c$ 共 $2c$ 个词。这样整体的迭代会更加的均衡。因为这个原因，Skip-Gram模型并没有和CBOW模型一样对输入进行迭代更新，而是对 $2c$ 个输出进行迭代更新。

这里总结下基于Hierarchical Softmax的Skip-Gram模型算法流程，梯度迭代使用了随机梯度上升法：

输入：基于Skip-Gram的语料训练样本，词向量的维度大小 M ，Skip-Gram的上下文大小 $2c$ ，步长 η

输出：霍夫曼树的内部节点模型参数 θ ，所有的词向量 w

1. 基于语料训练样本建立霍夫曼树。
2. 随机初始化所有的模型参数 θ ，所有的词向量 w ，
3. 进行梯度上升迭代过程，对于训练集中的每一个样本($w, context(w)$)做如下处理：

a) for $i = 1$ to $2c$:

i) $e=0$

ii) for $j = 2$ to l_w , 计算 :

$$f = \sigma(x_i^T \theta_{j-1}^i)$$

$$g = (1 - d_j^i - f)\eta$$

$$e = e + g\theta_{j-1}^i$$

$$\theta_{j-1}^i = \theta_{j-1}^i + gx_i$$

iii)

$$x_i = x_i + e$$

b)如果梯度收敛，则结束梯度迭代，算法结束，否则回到步骤a继续迭代。

5. Hierarchical Softmax的模型源码和算法的对应

这里给出上面算法和word2vec源码中的变量对应关系。

在源代码中，基于Hierarchical Softmax的CBOW模型算法在435-463行，基于Hierarchical Softmax的Skip-Gram的模型算法在495-519行。大家可以对着源代码再深入研究下算法。

在源代码中，neule对应我们上面的 e , syn0对应我们的 x_w , syn1对应我们的 θ_{j-1}^i , layer1_size对应词向量的维度，window对应我们的 C 。

另外，vocab[word].code[d]指的是，当前单词word的，第d个编码，编码不含Root结点。vocab[word].point[d]指的是，当前单词word，第d个编码下，前置的结点。

以上就是基于Hierarchical Softmax的word2vec模型，下一篇我们讨论基于Negative Sampling的word2vec模型。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

word2vec原理(三) 基于Negative Sampling的模型

[word2vec原理\(一\) CBOW与Skip-Gram模型基础](#)

[word2vec原理\(二\) 基于Hierarchical Softmax的模型](#)

word2vec原理(三) 基于Negative Sampling的模型

在上一篇中我们讲到了基于Hierarchical Softmax的word2vec模型，本文我们再来看看另一种求解word2vec模型的方法：Negative Sampling。

1. Hierarchical Softmax的缺点与改进

在讲基于Negative Sampling的word2vec模型前，我们先看看Hierarchical Softmax的缺点。的确，使用霍夫曼树来代替传统的神经网络，可以提高模型训练的效率。但是如果我们的训练样本里的中心词 w 是一个很生僻的词，那么就得在霍夫曼树中辛苦的向下走很久了。能不能不用搞这么复杂的一颗霍夫曼树，将模型变的更加简单呢？

Negative Sampling就是这么一种求解word2vec模型的方法，它摒弃了霍夫曼树，采用了Negative Sampling（负采样）的方法来求解，下面我们就来看看Negative Sampling的求解思路。

2. 基于Negative Sampling的模型概述

既然名字叫Negative Sampling（负采样），那么肯定使用了采样的方法。采样的方法有很多种，比如之前讲到的大名鼎鼎的MCMC。我们这里的Negative Sampling采样方法并没有MCMC那么复杂。

比如我们有一个训练样本，中心词是 w ，它周围上下文共有 $2c$ 个词，记为 $context(w)$ 。由于这个中心词 w ，的确和 $context(w)$ 相关存在，因此它是一个真实的正例。通过Negative Sampling采样，我们得到 neg 个和 w 不同的中心词 $w_i, i = 1, 2, \dots, neg$ ，这样 $context(w)$ 和 w_i 就组成了 neg 个并不真实存在的负例。利用这一个正例和 neg 个负例，我们进行二元逻辑回归，得到负采样对应每个词 w_i 对应的模型参数 $\theta_{\{i\}}$ ，和每个词的词向量。

从上面的描述可以看出，Negative Sampling由于没有采用霍夫曼树，每次只是通过采样 neg 个不同的中心词做负例，就可以训练模型，因此整个过程要比Hierarchical Softmax简单。

不过有两个问题还需要弄明白：1) 如果通过一个正例和 neg 个负例进行二元逻辑回归呢？2) 如何进行负采样呢？

我们在第三节讨论问题1，在第四节讨论问题2。

3. 基于Negative Sampling的模型梯度计算

Negative Sampling也是采用了二元逻辑回归来求解模型参数，通过负采样，我们得到了 neg 个负例 $(context(w), w_i) | i = 1, 2, \dots, neg$ 。为了统一描述，我们将正例定义为 w_0 。

在逻辑回归中，我们的正例应该期望满足：

$$P(context(w_0), w_i) = \sigma(x_{w_i}^T \theta^{w_0}), y_i = 1, i = 0$$

我们的负例期望满足：

$$P(context(w_0), w_i) = 1 - \sigma(x_i^T \theta^{w_0}), y_i = 0, i = 1, 2, \dots, neg$$

我们期望可以最大化下式：

$$\prod_{i=0}^{neg} P(context(w_0), w_i) = \sigma(x_{w_0}^T \theta^{w_0}) \prod_{i=1}^{neg} (1 - \sigma(x_{w_i}^T \theta^{w_0}))$$

利用逻辑回归和上一节的知识，我们容易写出此时模型的似然函数为：

$$\prod_{i=0}^{neg} \sigma(x_{w_i}^T \theta^{w_0})^{y_i} (1 - \sigma(x_{w_i}^T \theta^{w_0}))^{1-y_i}$$

此时对应的对数似然函数为：

$$L = \sum_{i=0}^{neg} y_i \log(\sigma(x_{w_i}^T \theta^{w_0})) + (1 - y_i) \log(1 - \sigma(x_{w_i}^T \theta^{w_0}))$$

和Hierarchical Softmax类似，我们采用随机梯度上升法，仅仅每次只用一个样本更新梯度，来进行迭代更新得到我们需要的 $x_{w_i}, \theta^{w_i}, i = 0, 1, \dots, neg$ ，这里我们需要求出 $x_{w_i}, \theta^{w_i}, i = 0, 1, \dots, neg$ 的梯度。

首先我们计算 θ^{w_i} 的梯度：

$$\begin{aligned}\frac{\partial L}{\partial \theta^{w_i}} &= y_i(1 - \sigma(x_{w_i}^T \theta^{w_i}))x_{w_i} - (1 - y_i)\sigma(x_{w_i}^T \theta^{w_i})x_{w_i} \quad (1) \\ &= (y_i - \sigma(x_{w_i}^T \theta^{w_i}))x_{w_i} \quad (2)\end{aligned}$$

同样的方法，我们可以求出 x_{w_i} 的梯度如下：

$$\frac{\partial L}{\partial x_{w_i}} = (y_i - \sigma(x_{w_i}^T \theta^{w_i}))\theta^{w_i}$$

有了梯度表达式，我们就可以用梯度上升法进行迭代来一步步的求解我们需要的 $x_{w_i}, \theta^{w_i}, i = 0, 1, \dots, neg$ 。

4. Negative Sampling负采样方法

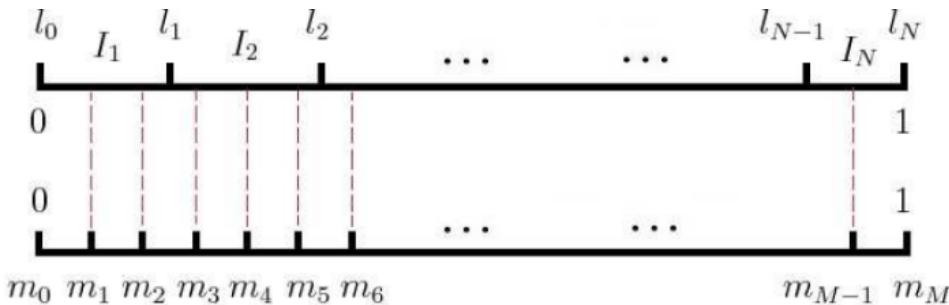
现在我们来看看如何进行负采样，得到 neg 个负例。word2vec采样的方法并不复杂，如果词汇表的大小为 V ，那么我们就将一段长度为1的线段分成 V 份，每份对应词汇表中的一个词。当然每个词对应的线段长度是不一样的，高频词对应的线段长，低频词对应的线段短。每个词 w 的线段长度由下式决定：

$$len(w) = \frac{count(w)}{\sum_{u \in vocab} count(u)}$$

在word2vec中，分子和分母都取了 $3/4$ 次幂如下：

$$len(w) = \frac{count(w)^{3/4}}{\sum_{u \in vocab} count(u)^{3/4}}$$

在采样前，我们将这段长度为1的线段划分成 M 等份，这里 $M >> V$ ，这样可以保证每个词对应的线段都会划分成对应的小块。而 M 份中的每一份都会落在某一个词对应的线段上。在采样的时候，我们只需要从 M 个位置中采样出 neg 个位置就行，此时采样到的每一个位置对应到的线段所属的词就是我们的负例词。



在word2vec中， M 取值默认为 10^8 。

5. 基于Negative Sampling的CBOW模型

有了上面Negative Sampling负采样的方法和逻辑回归求解模型参数的方法，我们就可以总结出基于Negative Sampling的CBOW模型算法流程了。梯度迭代过程使用了随机梯度上升法：

输入：基于CBOW的语料训练样本，词向量的维度大小 M ，CBOW的上下文大小 $2c$ ，步长 η ，负采样的个数 neg

输出：词汇表每个词对应的模型参数 θ ，所有的词向量 x_w

1. 随机初始化所有的模型参数 θ ，所有的词向量 w
2. 对于每个训练样本 $(context(w_0), w_0)$ ，负采样出 neg 个负例中心词 $w_i, i = 1, 2, \dots, neg$
3. 进行梯度上升迭代过程，对于训练集中的每一个样本 $(context(w_0), w_0, w_1, \dots, w_{neg})$ 做如下处理：

a) $e = 0$ ，计算 $x_{w_0} = \frac{1}{2c} \sum_{i=1}^{2c} x_i$

b) for $i = 0$ to neg ，计算：

$$f = \sigma(x_{w_i}^T \theta^{w_i})$$

$$g = (y_i - f)\eta$$

$$e = e + g\theta^{w_i}$$

$$\theta^{w_i} = \theta^{w_i} + gx_{w_i}$$

c) 对于 $context(w)$ 中的每一个词向量 x_j (共 $2c$ 个)进行更新：

$$x_j = x_j + e$$

d) 如果梯度收敛，则结束梯度迭代，否则回到步骤3继续迭代。

6. 基于Negative Sampling的Skip-Gram模型

有了上一节CBOW的基础和上一篇基于Hierarchical Softmax的Skip-Gram模型基础，我们也可以总结出基于Negative Sampling的Skip-Gram模型算法流程了。梯度迭代过程使用了随机梯度上升法：

输入：基于Skip-Gram的语料训练样本，词向量的维度大小 M ，Skip-Gram的上下文大小 $2c$ ，步长 η ，负采样的个数 neg 。

输出：词汇表每个词对应的模型参数 θ ，所有的词向量 x_w

1. 随机初始化所有的模型参数 θ ，所有的词向量 w

2. 对于每个训练样本($context(w_0), w_0$)，负采样出 neg 个负例中心词 $w_i, i = 1, 2, \dots, neg$

3. 进行梯度上升迭代过程，对于训练集中的每一个样本($context(w_0), w_0, w_1, \dots, w_{neg}$)做如下处理：

a) for i = 1 to 2c:

i) $e=0$

ii) for i= 0 to neg, 计算：

$$f = \sigma(x_{w_i}^T \theta^{w_i})$$

$$g = (y_i - f)\eta$$

$$e = e + g\theta^{w_i}$$

$$\theta^{w_i} = \theta^{w_i} + gx_{w_i}$$

iii) 对于 $context(w)$ 中的每一个词向量 x_j (共 $2c$ 个)进行更新：

$$x_j = x_j + e$$

b)如果梯度收敛，则结束梯度迭代，算法结束，否则回到步骤a继续迭代。

7. Negative Sampling的模型源码和算法的对应

这里给出上面算法和word2vec源码中的变量对应关系。

在源代码中，基于Negative Sampling的CBOW模型算法在464-494行，基于Hierarchical Softmax的Skip-Gram的模型算法在520-542行。大家可以对着源代码再深入研究下算法。

在源代码中，neule对应我们上面的 e ，syn0对应我们的 x_w ，syn1neg对应我们的 θ^{w_i} ，layer1_size对应词向量的维度，window对应我们的 c 。negative对应我们的 neg ，table_size对应我们负采样的划分数 M 。

另外，vocab[word].code[d]指的是，当前单词word的，第d个编码，编码不含Root结点。vocab[word].point[d]指的是，当前单词word，第d个编码下，前置的结点。这些和基于Hierarchical Softmax的是一样的。

以上就是基于Negative Sampling的word2vec模型，希望可以帮助到大家，后面会讲解用gensim的python版word2vec来使用word2vec解决实际问题。

(欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com)

用gensim学习word2vec

在word2vec原理篇中，我们对word2vec的两种模型CBOW和Skip-Gram，以及两种解法Hierarchical Softmax和Negative Sampling做了总结。这里我们就从实践的角度，使用gensim来学习word2vec。

1. gensim安装与概述

gensim是一个很好用的Python NLP的包，不光可以用于使用word2vec，还有很多其他的API可以用。它封装了google的C语言版的word2vec。当然我们可以直接使用C语言版的word2vec来学习，但是个人认为没有gensim的python版来的方便。

安装gensim是很容易的，使用"pip install gensim"即可。但是需要注意的是gensim对numpy的版本有要求，所以安装过程中可能会偷偷的升级你的numpy版本。而windows版的numpy直接装或者升级是有问题的。此时我们需要卸载numpy，并重新下载带mkl的符合gensim版本要求的numpy，下载地址在此：
<http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy>。安装方法和[scikit-learn 和pandas 基于windows单机机器学习环境的搭建](#)这一篇第4步的方法一样。

安装成功的标志是你可以在代码里做下面的import而不出错：

```
from gensim.models import word2vec
```

2. gensim word2vec API概述

在gensim中，word2vec 相关的API都在包gensim.models.word2vec中。和算法有关的参数都在类gensim.models.word2vec.Word2Vec中。算法需要注意的参数有：

1) sentences: 我们要分析的语料，可以是一个列表，或者从文件中遍历读出。后面我们会从文件读出的例子。

2) size: 词向量的维度，默认值是100。这个维度的取值一般与我们的语料的大小相关，如果是不大的语料，比如小于100M的文本语料，则使用默认值一般就可以了。如果是超大的语料，建议增大维度。

3) window : 即词向量上下文最大距离，这个参数在我们的算法原理篇中标记为 c ，window越大，则和某一词较远的词也会产生上下文关系。默认值为5。在实际使用中，可以根据实际的需求来动态调整这个window的大小。如果是小语料则这个值可以设的更小。对于一般的语料这个值推荐在[5,10]之间。

4) sg: 即我们的word2vec两个模型的选择了。如果是0，则是CBOW模型，是1则是Skip-Gram模型，默认是0即CBOW模型。

5) hs: 即我们的word2vec两个解法的选择了，如果是0，则是Negative Sampling，是1的话并且负采样个数negative大于0，则是Hierarchical Softmax。默认是0即Negative Sampling。

6) negative: 即使用Negative Sampling时负采样的个数，默认是5。推荐在[3,10]之间。这个参数在我们的算法原理篇中标记为neg。

7) cbow_mean: 仅用于CBOW在做投影的时候，为0，则算法中的 x_w 为上下文的词向量之和，为1则为上下文的词向量的平均值。在我们的原理篇中，是按照词向量的平均值来描述的。个人比较喜欢用平均值来表示 x_w ，默认值也是1，不推荐修改默认值。

8) min_count: 需要计算词向量的最小词频。这个值可以去掉一些很生僻的低频词，默认是5。如果是小语料，可以调低这个值。

9) iter: 随机梯度下降法中迭代的最大次数，默认是5。对于大语料，可以增大这个值。

10) alpha: 在随机梯度下降法中迭代的初始步长。算法原理篇中标记为 η ，默认是0.025。

11) min_alpha: 由于算法支持在迭代的过程中逐渐减小步长，min_alpha给出了最小的迭代步长值。随机梯度下降每轮的迭代步长可以由iter, alpha, min_alpha一起得出。这部分由于不是word2vec算法的核心内容，因此在原理篇我们没有提到。对于大语料，需要对alpha, min_alpha, iter一起调参，来选择合适的三个值。

以上就是gensim word2vec的主要的参数，下面我们用一个实际的例子来学习word2vec。

3. gensim word2vec实战

我选择的《人民的名义》的小说原文作为语料，语料原文在[这里](#)。

拿到了原文，我们首先要进行分词，这里使用结巴分词完成。在中文文本挖掘预处理流程总结中，我们已经对分词的原理和实践做了总结。因此，这里直接给出分词的代码，分词的结果，我们放到另一个文件中。代码如下，加入下面的一串人名是为了结巴分词能更准确的把人名分出来。

```
# -*- coding: utf-8 -*-

import jieba
import jieba.analyse

jieba.suggest_freq('沙瑞金', True)
jieba.suggest_freq('田国富', True)
jieba.suggest_freq('高育良', True)
jieba.suggest_freq('侯亮平', True)
jieba.suggest_freq('钟小艾', True)
jieba.suggest_freq('陈岩石', True)
jieba.suggest_freq('欧阳菁', True)
jieba.suggest_freq('易学习', True)
jieba.suggest_freq('王大路', True)
jieba.suggest_freq('蔡成功', True)
jieba.suggest_freq('孙连城', True)
jieba.suggest_freq('季昌明', True)
jieba.suggest_freq('丁义珍', True)
jieba.suggest_freq('郑西坡', True)
jieba.suggest_freq('赵东来', True)
jieba.suggest_freq('高小琴', True)
jieba.suggest_freq('赵瑞龙', True)
jieba.suggest_freq('林华华', True)
jieba.suggest_freq('陆亦可', True)
jieba.suggest_freq('刘新建', True)
jieba.suggest_freq('刘庆祝', True)

with open('./in_the_name_of_people.txt') as f:
    document = f.read()

#document_decode = document.decode('GBK')

document_cut = jieba.cut(document)
#print ' '.join(jieba_cut) //如果打印结果，则分词效果消失，后面的result无法显示
result = ' '.join(document_cut)
result = result.encode('utf-8')
with open('./in_the_name_of_people_segment.txt', 'w') as f2:
    f2.write(result)
f.close()
f2.close()
```

拿到了分词后的文件，在一般的NLP处理中，会需要去停用词。由于word2vec的算法依赖于上下文，而上下文有可能就是停词。因此对于word2vec，我们可以不用去停词。

现在我们可以直接读分词后的文件到内存。这里使用了word2vec提供的LineSentence类来读文件，然后套用word2vec的模型。这里只是一个示例，因此省去了调参的步骤，实际使用的时候，你可能需要对我们上面提到一些参数进行调参。

```
# import modules & set up logging
import logging
import os
from gensim.models import word2vec

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

sentences = word2vec.LineSentence('./in_the_name_of_people_segment.txt')

model = word2vec.Word2Vec(sentences, hs=1,min_count=1>window=3,size=100)
```

模型出来了，我们可以用来做什么呢？这里给出三个常用的应用。

第一个是最常用的，找出某一个词向量最相近的词集合，代码如下：

```
req_count = 5
for key in model.wv.similar_by_word('沙瑞金'.decode('utf-8'), topn=100):
    if len(key[0]) == 3:
        req_count -= 1
        print key[0], key[1]
    if req_count == 0:
        break;
```

我们看看沙书记最近的一些3个字的词（主要是人名）如下：

高育良 0.967257142067
李达康 0.959131598473
田国富 0.953414440155
易学习 0.943500876427
祁同伟 0.942932963371

第二个应用是看两个词向量的相近程度，这里给出了书中两组人的相似程度：

```
print model.wv.similarity('沙瑞金'.decode('utf-8'), '高育良'.decode('utf-8'))
print model.wv.similarity('李达康'.decode('utf-8'), '王大路'.decode('utf-8'))
```

输出如下：

0.961137455325
0.935589365706

第三个应用是找出不同类的词，这里给出了人物分类题：

```
print model.wv.doesnt_match(u"沙瑞金 高育良 李达康 刘庆祝".split())
```

word2vec也完成的很好，输出为“刘庆祝”。

以上就是用gensim学习word2vec实战的所有内容，希望对大家有所帮助。

（欢迎转载，转载请注明出处。欢迎沟通交流：pinard.liu@ericsson.com）