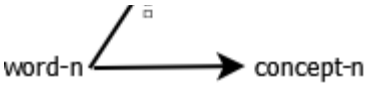
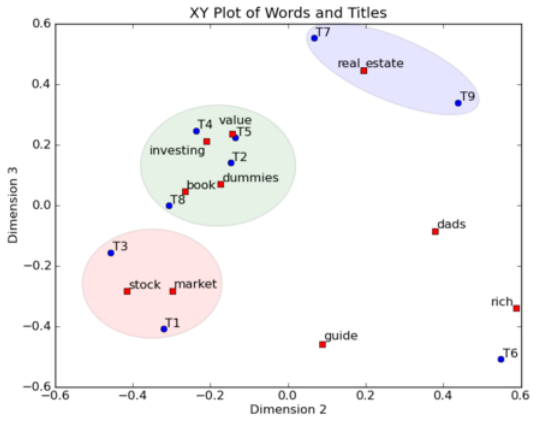
LSA，pLSA原理及其代码实现 - 笨兔勿应 - 博客园



LSA 的核心思想是将词和文档映射到**潜在语义空间**，再比较其相似性。
举个简单的栗子，对一个 Term-Document 矩阵做SVD分解，并将左奇异向量和右奇异向量都取后2维（之前是3维的矩阵），投影到一个平面上（潜在语义空间），可以得到：



在图上，每一个红色的点，都表示一个词，每一个蓝色的点，都表示一篇文档，这样我们可以对这些词和文档进行聚类，比如说 stock 和 market 可以放在一类，因为他们老是出现在一起，real 和 estate 可以放在一类，dads，guide 这种词就看起来有点孤立了，我们就不对他们进行合并了。按这样聚类出现的效果，可以提取文档集合中的近义词，这样当用户检索文档的时候，是用语义级别（近义词集合）去检索了，而不是之前的词的级别。这样一减少我们的检索、存储量，因为这样压缩的文档集合和PCA是异曲同工的，二可以提高我们的用户体验，用户输入一个词，我们可以在这个词的近义词的集合中去找，这是传统的索引无法做到的。

2. LSA的优点

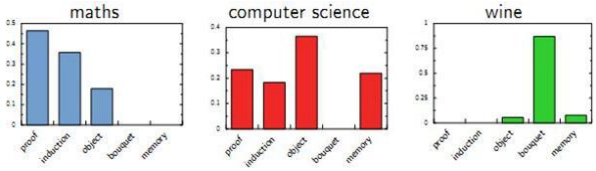
1）低维空间表示可以刻画同义词，同义词会对应着相同或相似的主题。
2）降维可去除部分噪声，是特征更鲁棒。
3）充分利用冗余数据。
4）无监督/完全自动化。
5）与语言无关。

3. LSA的缺点

1）LSA可以处理向量空间模型无法解决的一义多词(synonymy)问题，但不能解决**一词多义**(polysemy)问题。因为LSA将每一个词映射为潜在语义空间中的一个点，也就是说一个词的多个意思在空间中对于的是同一个点，并没有被区分。
2）SVD的优化目标基于L-2 norm 或者 Frobenius Norm 的，这相当于隐含了对数据的高斯分布假设。而 term 出现的次数是非负的，这明显不符合 Gaussian 假设，而更接近 Multi-nomial 分布。
3）特征向量的方向没有对应的物理解释。
4）SVD的计算复杂度很高，而且当有新的文档来到时，若要更新模型需重新训练。
5）没有刻画term出现次数的概率模型。
6）对于count vectors 而言，欧式距离表达是不合适的（重建时会产生负数）。
7）维数的选择是ad-hoc的。

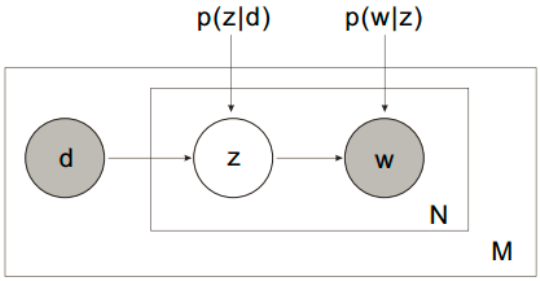
二. pLSA

首先，我们可以看看日常生活中人是如何构思文章的。如果我们要写一篇文章，往往是先确定要写哪几个主题。譬如构思一篇自然语言处理相关的文章，可能40%会谈论语言学，30%谈论概率统计，20%谈论计算机，还有10%谈论其它主题。
对于语言学，容易想到的词包括：语法，句子，主语等；对于概率统计，容易想到的词包括：概率，模型，均值等；对于计算机，容易想到的词包括：内存，硬盘，编程等。我们之所以能想到这些词，是因为这些词在对应的主题下出现的概率很高。我们可以很自然的看到，一篇文章通常是由多个主题构成的，而每一个主题大概可以用与该主题相关的频率最高的一些词来描述。以上这种想法由Hofmann于1999年给出的pLSA模型中首先进行了明确的数学化。Hofmann认为一篇文章（Doc）可以由多个主题（Topic）混合而成，而每个Topic都是词汇上的概率分布，文章中的每个词都是由一个固定的Topic生成的。下图是英语中几个Topic的例子。



pLSA的建模思路分为两种。

1. 第一种思路



以$p(d_m)$的概率从文档集合D中选择一个文档d_m

以$p(z_k|d_m)$的概率从主题集合Z中选择一个主题z_k

以$p(w_n|z_k)$的概率从词集W中选择一个词w_n

有几点说明：

- 以上变量有两种状态：observed (d_m & w_n) 和 latent (z_k)
- W来自文档，但同时是集合**（元素不重复）**，相当于一个词汇表

直接的，针对observed variables做建立likelihood function：

$$\begin{aligned} L &= \Pi_m^M \Pi_n^N p(d_m, w_n) \\ &= \Pi_m^{M'} \Pi_n^{N'} p(d_m, w_n)^{n(d_m, w_n)} \end{aligned}$$

其中，$n(d_m, w_n)$为(d_m, w_n)pair出现的次数。为加以区分，之后使用M'与N'标识对应文档与词汇数量。两边取$\log$，得：

$$\begin{aligned} l &= \sum_m^{M'} \sum_n^{N'} n(d_m, w_n) \log p(d_m, w_n) \\ &= \sum_m^{M'} \sum_n^{N'} n(d_m, w_n) \log p(d_m) p(w_n|d_m) \qquad (*) \end{aligned}$$

$$= \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \log p(d_m) \sum_{k}^{K} p(z_k|d_m) p(w_n|z_k)$$
$$= \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \log \sum_{k}^{K} p(d_m) p(z_k|d_m) p(w_n|z_k)$$

其中，倒数第二步旨在将z_k暴露出来。由于likelihood function中$p(z_k|d_m)$与$p(w_n|z_k)$存在latent variable，难以直接使用MLE求解，很自然想到用E-M算法求解。E-M算法主要分为Expectation与Maximization两步。

Step 1: Expectation

假设已知$p(z_k|d_m)$与$p(w_n|z_k)$，求latent variablez_k的后验概率$p(z_k|d_m, w_n)$

$$\begin{aligned} p(z_k|d_m, w_n) &= \frac{p(z_k, d_m, w_n)}{p(d_m, w_n)} \\ &= \frac{p(z_k, d_m, w_n)}{p(d_m, w_n)} \\ &= \frac{p(z_k|d_m) p(w_n|z_k)}{p(w_n|d_m)} \\ &= \frac{p(z_k|d_m) p(w_n|z_k)}{\sum_{k'}^{K} p(z_{k'}|d_m) p(w_n|z_{k'})} \end{aligned}$$

Step 2: Maximization

求关于参数$p(z_k|d_m)$和$p(w_n|z_k)$的**Complete data对数似然函数期望**的极大值，得到最优解。带入E步迭代循环。

由$(*)$式可得：

$$l = \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \log p(w_n|d_m) + \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \log p(d_m)$$

此式后部分为常量。故令：

$$l' = \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \log p(w_n|d_m)$$
$$\begin{aligned} E(l') &= \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \sum_{k}^{K} p(z_k|d_m, w_n) \log p(w_n, z_k|d_m) \\ &= \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \sum_{k}^{K} p(z_k|d_m, w_n) \log p(w_n|z_k) p(z_k|d_m) \end{aligned}$$

建立以下目标函数与约束条件：

$$\begin{aligned} E(l') &= \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \sum_{k}^{K} p(z_k|d_m, w_n) \log p(w_n|z_k) p(z_k|d_m) \\ s.t. \quad & \sum_{k}^{K} p(z_k|d_m) = 1 \\ & \sum_{n}^{N} p(w_n|z_k) = 1 \end{aligned}$$

只有等式约束，使用Lagrange乘子法解决：

$$f = \sum_{m}^{M'} \sum_{n}^{N'} n(d_m, w_n) \sum_{k}^{K} p(z_k|d_m, w_n) \log p(w_n|z_k) p(z_k|d_m) + \sum_{m}^{M} \lambda_m (1 - \sum_{k}^{K} p(z_k|d_m)) + \sum_{k}^{K} \lambda_k (1 - \sum_{n}^{N} p(w_n|z_k))$$

对$p(z_k|d_m)$与$p(w_n|z_k)$求驻点，得：

$$\frac{\partial f}{\partial p(z_k|d_m)} = \frac{\sum_{n}^{N'} n(d_m, w_n) p(z_k|d_m, w_n)}{p(z_k|d_m)} - \lambda_m$$

$$\frac{\partial f}{\partial p(w_n|z_k)} = \frac{\sum_m^M n(d_m, w_n)p(z_k|d_m, w_n)}{p(w_n|z_k)} - \lambda_k$$

令$\frac{\partial f}{\partial p(z_k|d_m)} = 0$，得：

$\lambda_m = \sum_k^K \sum_n^{N'} n(d_m, w_n)p(z_k|d_m, w_n)$，故有：

$$p(z_k|d_m) = \frac{\sum_n^{N'} n(d_m, w_n)p(z_k|d_m, w_n)}{\sum_k^K \sum_n^{N'} n(d_m, w_n)p(z_k|d_m, w_n)}$$

同理，有：

$$p(w_n|z_k) = \frac{\sum_m^{M'} n(d_m, w_n)p(z_k|d_m, w_n)}{\sum_n^N \sum_m^{M'} n(d_m, w_n)p(z_k|d_m, w_n)}$$

将$p(z_k|d_m)$与$p(w_n|z_k)$回代Expectation：

$p(z_k|d_m, w_n) = \frac{p(z_k|d_m)p(w_n|z_k)}{\sum_{k'}^K p(z_k'|d_m)p(w_n|z_k')}$，循环迭代。

pLSA的建模思想较为简单，对于observed variables建立likelihood function，将latent variable暴露出来，并使用E-M算法求解。其中M步的标准做法是引入Lagrange乘子求解后回代到E步。

总结一下使用EM算法求解pLSA的基本实现方法：

(1)E步骤：求隐含变量Given当前估计的参数条件下的后验概率。
(2)M步骤：最大化**Complete data对数似然函数**的期望，此时我们使用E步骤里计算的隐含变量的后验概率，得到新的参数值。
两步迭代进行直到收敛。

2. 第二种思路

这个思路和上面思路的区别就在于对P(d,w)的展开公式使用的不同，思路二使用的是3个概率来展开的，如下：

$$P(d, w) = \sum_{z \in Z} P(z)P(d|z)P(w|z)$$

这样子我们后面的EM算法的大致思路都是相同的，就是表达形式变化了，最后得到的EM步骤的更新公式也变化了。当然，思路二得到的是3个参数的更新公式。如下：

- E步骤(求期望)：

$$P(z|d, w) = \frac{P(z)P(d|z)P(w|z)}{\sum_{z'} P(z')P(d|z')P(w|z')},$$

- M步骤(使似然函数最大)：

$$P(w|z) = \frac{\sum_d n(d, w)P(z|d, w)}{\sum_{d,w'} n(d, w')P(z|d, w')},$$

$$P(d|z) = \frac{\sum_w n(d, w)P(z|d, w)}{\sum_{d',w} n(d', w)P(z|d', w)},$$

$$P(z) = \frac{1}{R}\sum n(d, w)P(z|d, w), \ R \equiv \sum n(d, w).$$

你会发现，其实多了一个参数是P(z)，参数P（d|z）变化了（之前是P(z|d)），然后P(w|z)是不变的，计算公式也相同。

计算公式也相同。

给定一个文档d，我们可以将其分类到一些主题词类别下。

PLSA算法可以通过训练样本的学习得到三个概率，而对于一个测试样本，其中P(w|z)概率是不变的，但是P(z)和P(d|z)都是需要重新更新的，我们也可以使用上面的EM算法，假如测试样本d的数据，我们可以得到新学习的P(z)和P(d|z)参数。这样我们就可以计算：

$$p(z/d)=\frac{p(z)p(d/z)}{\sum_{z\in Z}p(d/z)p(z)}$$

为什么要计算P(z|d)呢？因为给定了一个测试样本d，要判断它是属于那些主题的，我们就需要计算P(z|d)，就是给定d，其在主题z下成立的概率是多少，不就是要计算 $p(z/d)$ 吗。这样我们就可以计算文档d在所有主题下的概率了。

这样既可以把一个测试文档划归到一个主题下，也可以给训练文档打上主题的标记，因为我们也是可以计算训练文档它们的的 $p(z/d)$ 。如果从这个应用思路来说，思路一说似乎更加直接，因为其直接计算出来了 $p(z/d)$ 。

3. pLSA的优势

1）定义了概率模型，而且每个变量以及相应的概率分布和条件概率分布都有明确的物理解释。
2）相比于LSA隐含了高斯分布假设，pLSA隐含的Multi-nomial分布假设更符合文本特性。
3）pLSA的优化目标是是KL-divergence最小，而不是依赖于最小均方误差等准则。
4）可以利用各种model selection和complexity control准则来确定topic的维数。

4. pLSA的不足

1）概率模型不够完备：在document层面上没有提供合适的概率模型，使得pLSA并不是完备的生成式模型，而必须在确定document i的情况下才能对模型进行随机抽样。
2）随着document和term 个数的增加，pLSA模型也线性增加，变得越来越庞大。
3）EM算法需要反复的迭代，需要很大计算量。

针对pLSA的不足，研究者们又提出了各种各样的topic based model, 其中包括大名鼎鼎的Latent Dirichlet Allocation (LDA)。

三. pLSA的Python代码实现

1. preprocess.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import numpy as np
class Preprocess:
    def __init__(self, fname, fsw):
        self.fname = fname
        # doc info
```

```
        self.docs = []
        self.doc_size = 0
        # stop word info
        self.sws = []
        # word info
        self.w2id = {}
        self.id2w = {}
        self.w_size = 0
        # stop word list init
        with open(fsw, 'r') as f:
            for line in f:
                self.sws.append(line.strip())
    def __work(self):
        with open(self.fname, 'r') as f:
            for line in f:
                line_strip = line.strip()
                self.doc_size += 1
                self.docs.append(line_strip)
                items = line_strip.split()
                for it in items:
                    if it not in self.sws:
                        if it not in self.w2id:
                            self.w2id[it] = self.w_size
                            self.id2w[self.w_size] = it
                            self.w_size += 1
        self.w_d = np.zeros([self.w_size, self.doc_size], dtype=np.int)
        for did, doc in enumerate(self.docs):
            ws = doc.split()
            for w in ws:
                if w in self.w2id:
                    self.w_d[self.w2id[w]][did] += 1
    def get_w_d(self):
        self.__work()
        return self.w_d
    def get_word(self, wid):
        return self.id2w[wid]
if __name__ == '__main__':
    fname = './data.txt'
    fsw = './stopwords.txt'
    pp = Preprocess(fname, fsw)
```

2. plsa.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import numpy as np
import time
import logging
def normalize(vec):
    s = sum(vec)
    for i in range(len(vec)):
        vec[i] = vec[i] * 1.0 / s
```

```
def llhood(w_d, p_z, p_w_z, p_d_z):
    V, D = w_d.shape
    ret = 0.0
    for w, d in zip(*w_d.nonzero()):
        p_d_w = np.sum(p_z * p_w_z[w,:] * p_d_z[d,:])
        if p_d_w > 0:
            ret += w_d[w][d] * np.log(p_d_w)
    return ret
class PLSA:
    def __init__(self):
        pass
    def train(self, w_d, Z, eps):
        V, D = w_d.shape
        # create prob array, p(d|z), p(w|z), p(z)
        p_d_z = np.zeros([D, Z], dtype=np.float)
        p_w_z = np.zeros([D, Z], dtype=np.float)
        p_z = np.zeros([Z], dtype=np.float)
        # initialize
        p_d_z = np.random.random([D, Z])
        for d_idx in range(D):
            normalize(p_d_z[d_idx])
        p_w_z = np.random.random([V, Z])
        for w_idx in range(V):
            normalize(p_w_z[w_idx])
        p_z = np.random.random([Z])
        normalize(p_z)
        # iteration until converge
        step = 1
        pp_d_z = p_d_z.copy()
        pp_w_z = p_w_z.copy()
        pp_z = p_z.copy()
        while True:
            logging.info('[ iteration ] step %d' % step)
            step += 1
            p_d_z *= 0.0
            p_w_z *= 0.0
            p_z *= 0.0
            # run EM algorithm
            for w_idx, d_idx in zip(*w_d.nonzero()):
                #print '[ EM ] >>>>>> E step : '
                p_z_d_w = pp_z * pp_d_z[d_idx,:] * pp_w_z[w_idx,:]
                normalize(p_z_d_w)
                #print '[ EM ] >>>>>> M step : '
                tt = w_d[w_idx, d_idx] * p_z_d_w
                p_w_z[w_idx,:] += tt
                p_d_z[d_idx,:] += tt
                p_z += tt
            normalize(p_w_z)
            normalize(p_d_z)
            p_z = p_z / w_d.sum()
            # check converge
            l1 = llhood(w_d, pp_z, pp_w_z, pp_d_z)
            l2 = llhood(w_d, p_z, p_w_z, p_d_z)
            diff = l2 - l1
            logging.info('[ iteration ] l2-l1  %.3f - %.3f = %.3f ' % (l2, l1, diff))
```

```
            if abs(diff) < eps:
                logging.info('[ iteration ] End EM ')
                return (l2, p_d_z, p_w_z, p_z)
            pp_d_z = p_d_z.copy()
            pp_w_z = p_w_z.copy()
            pp_z = p_z.copy()
```

3. main.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from preprocess import Preprocess as PP
from plsa import PLSA
import numpy as np
import logging
import time
def main():
    # setup logging --------------------------
    logging.basicConfig(filename='plsa.log',
                        level=logging.INFO,
                        format='%(asctime)s %(filename)s[line:%(lineno)d] %
(levelname)s %(message)s',
                        datefmt='%a, %d %b %Y %H:%M:%S')
    #console = logging.StreamHandler()
    #console.setLevel(logging.INFO)
    #logging.getLogger('').addHandler(console)
    # some basic configuration ---------------
    fname = './data.txt'
    fsw = './stopwords.txt'
    eps = 20.0
    key_word_size = 10
    # preprocess -----------------------------
    pp = PP(fname, fsw)
    w_d = pp.get_w_d()
    V, D = w_d.shape
    logging.info('V = %d, D = %d' % (V, D))
    # train model and get result -------------
    pmodel = PLSA()
    for z in range(3, (D+1), 10):
        t1 = time.clock()
        (l, p_d_z, p_w_z, p_z) = pmodel.train(w_d, z, eps)
        t2 = time.clock()
        logging.info('z = %d, eps = %f, time = %f' % (z, l, t2-t1))
        for itz in range(z):
            logging.info('Topic %d' % itz)
            data = [(p_w_z[i][itz], i) for i in range(len(p_w_z[:,itz]))]
            data.sort(key=lambda tup:tup[0], reverse=True)
            for i in range(key_word_size):
                logging.info('%s : %.6f ' % (pp.get_word(data[i][1]), data[i][0]))
if __name__ == '__main__':
    main()
```

版权声明:

分类: Natural Language Processing

标签: NLP, LSA, pLSA, EM

笨兔勿应
关注 - 4
粉丝 - 60
+加关注

1 0

» 下一篇: Click Models for Web Search(1) - Basic Click Models

posted @ 2016-12-25 17:24 笨兔勿应 阅读(13001) 评论(0) 编辑 收藏