LDA.py の解説

木村　健

平成 30 年 6 月 27 日

1　このドキュメントについて

本ドキュメントは LDA(Latent Dirichret Allocation) について基礎的な知識を与えると同時に持橋先生が作成された LDA.py のコードについて解説を与える。

ドキュメントの読者は Python に詳しいことと基本的な確率・統計、線形代数、微分積分の知識を有することを前提としているが全部わからなくても読めると思う。

2　LDA のタスクと入力パラメータ

LDA[2] はドキュメント群（ここでは簡単のため英語とする）の各単語について潜在的トピックを推定する技法である。いくつのトピックに分けるかは最初に指定する。常に同じ順序でトピックが抽出されると限らない。unsupervised learning である（教師値を用いない)。

LDA の開始に先立って次のパラメータが必要となる (LDA.py の index.html より引用加筆) Gibbs sampling という技法を利用していることを付け加えておく。

- K: topics: number of topics in LDA
- N: iters: number of Gibbs iterations
- α: Dirichlet hyperparameter on topics
- β: Dirichlet hyperparameter on words

K がトピック数で、はじめに与える。例えば $K = 10$ とする。N はイテレーションの回数でこれを大きくするほど perplexity が改善（小さくなる）するがある程度まで行くとあまり動かなくなる。K が大きいほど収束しずらくなる。

3　LDA.py の大まかな流れ

3.1　引数からパラメータを計算

まず引数からパラメータ (K,N など) を計算する。同時に train ファイルとして入力ファイル名を特定するのと同時に出力ファイル群の prefix を決定する。

3.2　入力ファイルの処理

SVMLight の擬似形式のファイルを読み込む。各レコードが一つのドキュメントに相当する。レコードはフィールド群からなり、”単語番号:頻度”という形式をしている。これを単語番号だけの「擬似平文」を作り出し、同時に与えるトピックのリストのリストも作る。

3.3 Gibbs sampling への入力を決定する

大まかには各パラメータを決定し、必要なベクトル、行列を準備する。また Cython で書かれた部分の最初で初期化をするがこれは行列などにカウント値を入れる初期化をしている

3.4 Gibbs sampling

まず特定の単語について、各頻度行列から 1 だけ頻度を引く（なかったことにする）。これは更新の式が自分のトピックを除いた式だからである。そしてその単語に紐づいている古いトピック番号を新しいトピック番号に変えて（どれが新しいトピックかは確率値から決まる）頻度行列を +1 する（戻す）。これを全ての単語について行い、全て終わって 1 イテレーションとする。perplexity を表示しながらこれを iters 回数実行する。

3.5 出力データの出力

テキスト形式で各行列などをファイルに書き込む。

3.6 これで終わりです

プログラムを終了する。

4 LDA.py の簡単な解説（コードを巡る）

4.1 main

コードを追っていく。まず main から。

```
def main ():
    cls = '\x1b[K'
    shortopts = "K:N:a:b:h"
    longopts = ['topics=','iters=','alpha=','beta=','help']
    K      = 10
    iters  = 1
    alpha  = 0
    beta   = 0.01
```

K のデフォルト値は 10、iters(N) のデフォルト値は 1 である。index.html では K=10, N=100 を与えている。

alpha と beta はディリクレ分布のハイパーパラメータである。beta はほとんど指定することがないのでデフォルトの 0.01 である。alpha は 0 である場合に限り $50/K$ が与えられ、alpha=5 となる。alpha は topics のためのハイパーパラメータ、beta は words のためのハイパーパラメータである（後述）。

```
    if alpha == 0:
        alpha = 50.0 / K

    if len(args) == 2:
        train = args[0]
        model = args[1]
    else:
        usage ()
```

```
    eprint('LDA: K = %d, iters = %d, alpha = %g, beta = %g' \
           % (K,iters,alpha,beta))
    logging.start (sys.argv, model)
```

4.2 ldaload

一通り arguments を処理すると train ファイルの読み込みが始まる。この読み込みは ldadata 関数が担当する。

```
    eprintf('loading data.. ')
    W,Z = ldaload (train, K)
```

ldadata は小さな関数であるが意義は大きい。SVMLight 形式（もどき）になっているドキュメント単位の単語頻度情報を読み込み、randtopic で全ての単語に乱数でトピックを（暫定的に）与える。

```
def ldaload (file, K):
    words = fmatrix.plain (file)
    topics = randtopic (words, K)
    return int32(words), int32(topics)
```

4.2.1 fmatrix.plain

ここで fmatrix.plain について説明する前に train として想定している入力フォーマットについて述べる。一行を各ドキュメントとして、単語番号とその単語の頻度を":"で結合したフィールドが可変長続く。疎行列を表現するときによく用いられる手法である。

> word0:value0 word1:value1 ... wordX:valueX
> 例：1:10 5:12 ... 1010:2

つまり SVMLight の疎行列入力について、ラベル（教師値）が欠落している状態である。ただし $wid = 0$ を SVM は許容していないが、本フォーマットは許容している。このため相互運用においては注意が必要である。もうフォーマットは分かっているのでざっと見ていく。

```
import numpy as np

def plain (file):
    """
    build a plain word sequence for LDA.
    """
    data = []
    with open (file, 'r') as fh:
        for line in fh:
            words = parse (line)
            if len(words) > 0:
                data.append (words)
    return data

def parse (line):
    words = []
    tokens = line.split()
```

```
    if len(tokens) > 0:
        for token in tokens:
            [id,cnt] = token.split(':')
            # w = int(id) - 1
            w = int(id)
            c = int(cnt)
            words.extend ([w for x in xrange(c)])
        return words
    else:
        return []
```

最終的に返すのはリストのリスト、data である。ここで疎行列データは単語番号による「擬似平文」に変換される。つまりあるフィールド 10:4 があったとすると、これが [10,10,10,10] と展開される。単語番号の若い順に並んでいることが想定されるので（SVM の標準的なフォーマットではソートされた素性番号が使用される）、これは平文そのものではない。

data[i][j] について、i はドキュメントのインデックスであり、j は単語の位置を表すインデックスである。data[i][j] で得られるものは単語番号である。

こうして data は ldaload 関数で words として与えられる。

4.3 ldaload 再び

（再掲）

```
def ldaload (file, K):
    words = fmatrix.plain (file)
    topics = randtopic (words, K)
    return int32(words), int32(topics)

def randtopic (words, K):
    topics = []
    for word in words:
        topic = npr.randint(K, size=len(word))
        topics.append (topic)
    return topics
```

randtopic 関数は words の全ての要素を [0,K) のトピック番号で埋める。しかもランダムに埋める。なので seed を指定しないとトピックの初期分布は毎回異なる。

できた topics は words と同じ形、ただし中身トピック番号となる。words, topics のドキュメントごとの要素は plain な Python の array ではなく numpy.array を使う。つまり numpy.array の python array である。これは int32 関数（自前）でこの形に修正される。

4.4 登場人物勢揃い

```
    W,Z = ldaload (train, K)
    D   = len(W)
    V   = lexicon(W)
    N   = datalen(W)
    NW  = np.zeros ((V,K), dtype=np.int32)
    ND  = np.zeros ((D,K), dtype=np.int32)
    NWsum = np.zeros (K, dtype=np.int32)
```

```
    eprint ('documents = %d, lexicon = %d, nwords = %d' % (D,V,N))
```

ここでほぼ初期状態で必要なものが揃う。

W は words=[numpy.array(wid0, wid1, ... , wid_n), numpy.array(wid0, ...)] Z はランダムで作られた単語あたりの topics=[numpy.array(t0, t1, ..., t_n), numpy.array(t0,...)] となる。D は W の長さなのでドキュメントの数となる。

lexicon と datalen について説明を加えたい。

```
def lexicon (words):
    v = None
    for word in words:
        if max(word) > v:
            v = max(word)
    return v + 1

def datalen (W):
    n = 0
    for w in W:
        n += len(w)
    return n

def datalen (W):
    return sum (map (lambda x: x.shape[0], W))
```

datalen が二つ存在する。どちらが実行されているかは不明であるがおそらく numpy 版の方が実行される。やることは W の要素数を数えるだけである。

lexicon は単純な関数で、words 内の単語番号で一番大きいものに 1 を足して返す。これは words 内に単語番号 0 の単語があるということだろうか、微妙な誤差が出そうな気はする。

つまり $V = |V|$ 語彙数、$N =$ 要素数 $=$ 単語数 である。

```
    NW  = np.zeros ((V,K), dtype=np.int32)
    ND  = np.zeros ((D,K), dtype=np.int32)
    NWsum = np.zeros (K, dtype=np.int32)
    eprint ('documents = %d, lexicon = %d, nwords = %d' % (D,V,N))
```

もう少しで登場人物が揃う。

- NW $= R^{V \times K}$ （語彙数 x トピック数）
- ND $= R^{D \times K}$ （ドキュメント数 x トピック数）
- NWsum $= R^{K}$ （トピック数）

注意として、1 次元目が rows、2 次元目が columns である。

4.5 80/20% parts

こうしてメインルーチンの最も計算量が多いパートへと辿り着く。

```
    # initialize
    eprint('initializing..')
    ldac.init (W, Z, NW, ND, NWsum)

    for iter in xrange(iters):
```

```
        elprintf('Gibbs iteration [%2d/%2d] PPL = %.4f' \
                % (iter+1,iters,ldappl(W,Z,NW,ND,alpha,beta)))
        ldac.gibbs (W, Z, NW, ND, NWsum, alpha, beta)
    eprintf('\n')
```

ldac は冒頭で import されているがこれは Cython のモジュールで、正体はシェアードライブラリ化された ldac.so である。これが Cython モジュールの場合 Python のコードの代わりに実行される。Cython は Python に Type の概念が入っているシンタックスのファイルで、C とも連携しやすい形である。主な用途としては重い処理を高速化するために用いる。

やっていることは最初に ldac.init してそれから iters 回（Gibbs samplings の回数）だけループしながら毎回 perplexity を表示（ldappl 関数）した後で ldac.gibbs（Gibbs Sampling の正体）を呼んでいるだけである。

4.6 Gibbs sampling 初期化

こうしてメインルーチンの最も計算量が多いパートへと辿り着く。

```
def init (list W,
          list Z,
          np.ndarray[dtype_t,ndim=2] NW not None,
          np.ndarray[dtype_t,ndim=2] ND not None,
          np.ndarray[dtype_t,ndim=1] NWsum not None):
    cdef int D = ND.shape[0]
    cdef int N
    cdef Py_ssize_t w, z

    for n in range(D):
        N = len(Z[n])
        for i in range(N):
            w        = W[n][i]
            z        = Z[n][i]
            NW[w,z] += 1
            ND[n,z] += 1
            NWsum[z] += 1
```

D はドキュメント数である。ループが二重になっている。外側はドキュメント数のループ、inner loop はドキュメント n に含まれる単語数である。つまり W および Z の全ての要素をトラバースする。

inner loop 内で w と z として単語番号とトピック番号を W と Z より取り出す。まず $NW_{w,z}$ をインクリメントする。つまりこの行列にはどの語彙がどのトピックで何回登場したか、が入っている。

次に $ND_{n,z}$ をインクリメントする。ND はどのドキュメントで、どのトピックが、どのくらい登場したかのカウント値となる。

最後に $NWsum_z$ をインクリメントする。これはトピックの数だけ要素があるから、どのトピックが全体でどれだけ出たかのカウント値となる。

つまり最初に $NW, ND, NWsum$ を正しいカウント値に初期化する。

4.7 Gibbs sampling 本丸

```
def gibbs (list W,
           list Z,
           np.ndarray[dtype_t,ndim=2] NW not None,
```

```
        np.ndarray[dtype_t,ndim=2] ND not None,
        np.ndarray[dtype_t,ndim=1] NWsum not None,
        double alpha, double beta):
    cdef int D = ND.shape[0]
    cdef int K = ND.shape[1]
    cdef int V = NW.shape[0]
    cdef int N
    cdef np.ndarray[ftype_t,ndim=1] p = np.zeros (K)
    cdef np.ndarray[dtype_t,ndim=1] Wn
    cdef np.ndarray[dtype_t,ndim=1] Zn
    cdef Py_ssize_t n, i, j, k, Wni, z, znew
    cdef double total
    cdef np.ndarray[ftype_t,ndim=1] rands
    cdef np.ndarray[dtype_t,ndim=1] seq = np.zeros(D,dtype=np.int32)

    # shuffle
    for n in range(D):
        seq[n] = n
    npr.shuffle (seq)

    # sample
    for j in range(D):
        n  = seq[j]
        N  = len(Z[n])
        Zn = Z[n]
        Wn = W[n]
        rands = npr.rand (N)
        for i in range(N):
            Wni        = Wn[i]
            z          = Zn[i]
            NW[Wni,z] -= 1
            ND[n,z]   -= 1
            NWsum[z]  -= 1

            total = 0.0
            for k in range(K):
                p[k] = (NW[Wni,k] + beta) / (NWsum[k] + V * beta) * \
                       (ND[n,k] + alpha)
                total += p[k]

            rands[i] = total * rands[i]
            total = 0.0
            znew  = 0
            for k in range(K):
                total += p[k]
                if rands[i] < total:
                    znew = k
                    break

            Zn[i]         = znew
            NW[Wni,znew] += 1
            ND[n,znew]   += 1
            NWsum[znew]  += 1
```

・・・・・・・ながー～い・・・・・・・・

少しづつ見ていきたい。ここが核心である。アイデアとしては頻度行列の帳尻が合うように特定アイテムの頻度をデクリメントして、確率値より新しいトピック Zn を求めて、このトピックで頻度行列をインクリメントしている。つまり一個取り出して一個追加している。これを長い間やり続ければより確かなトピック分類に近く、と言うアイデアである。（アイデア自体は簡単）

4.8 Gibbs Sampling(1): 各変数おさらい

```
def gibbs (list W,
           list Z,
           np.ndarray[dtype_t,ndim=2] NW not None,
           np.ndarray[dtype_t,ndim=2] ND not None,
           np.ndarray[dtype_t,ndim=1] NWsum not None,
           double alpha, double beta):
    cdef int D = ND.shape[0]
    cdef int K = ND.shape[1]
    cdef int V = NW.shape[0]
    cdef int N
    cdef np.ndarray[ftype_t,ndim=1] p = np.zeros (K)
    cdef np.ndarray[dtype_t,ndim=1] Wn
    cdef np.ndarray[dtype_t,ndim=1] Zn
    cdef Py_ssize_t n, i, j, k, Wni, z, znew
    cdef double total
    cdef np.ndarray[ftype_t,ndim=1] rands
    cdef np.ndarray[dtype_t,ndim=1] seq = np.zeros(D,dtype=np.int32)
```

ここでは文献 [1] と照らし合わせながらパラメーターを復習したい。

コード内の名前	[1] 内での式	説明
W	y_{il}	ドキュメント i の l 番目単語の単語番号 y_{il} の入ったリストのリスト
Z	q_{il}	ドキュメント i の l 番目のトピック番号 q_{il} の入ったリストのリスト
	C_{ivk}	ドキュメント i、単語番号 v、トピック k のカウント値
NW(V x K)	$C_{vk} = \sum_{i=1}^{D} C_{ivk}$	単語 v でトピック k の頻度をドキュメントで周辺化した行列
ND(D x K)	$C_{ik} = \sum_{v=1}^{\|V\|} C_{ivk}$	ドキュメント i でトピック k を語彙で周辺化したカウント値の行列
NWsum(K)	$C_k = \sum_{v=1}^{\|V\|} C_{vk}$	トピック k の頻度をドキュメントと語彙で周辺化した配列
D	N	ドキュメント数
K	K	トピック数
V	$\|V\|$	語彙数
N	L_i	ドキュメント i の長さ (単語数)
p[k]		$p(q_{i,l} = k \| q_{-\ i,l}, y_i, \alpha, \gamma) = \frac{C_{(v=y_{il}),k} + \gamma}{C_k + \|V\|\gamma} \{C_{ik} + \alpha\}$
Wn	$y_{i,l=[1:L_i]}$	ドキュメント i の単語番号配列（長さ L_i)
Zn	$q_{i,l=[1:L_i]}$	ドキュメント i のトピック番号配列（長さ L_i)
n	i	シャッフルされたドキュメント群から抽出されたドキュメント番号 i
i	l(エル)	ドキュメント内の単語位置 l
j		シャッフルされる前のドキュメント番号 i
k	k	トピック番号 k
Wni	y_{il}	ドキュメント i、単語位置 l の単語番号 y_{il}
z	q_{il}	ドキュメント i、単語位置 l のトピック番号 q_{il}
znew	$\hat{q_{il}}$?	ドキュメント i 単語位置 l のより尤もらしいトピック番号
total(最初)		$= \sum_{k=1}^{K} p(q_{i,l} = k \| q_{-\ i,l}, y_i, \alpha, \gamma)$?
total(二番目)		$= \sum_{k'=1}^{k} p(k')$
rands		random samples from a uniform distribution over [0, 1).
rands（ループ内）		$rands(l) = rands(l) \sum_{k=1}^{K} p(q_{i,l} = k \| q_{-\ i,l}, y_i, \alpha, \gamma)$
seq		$0, 1, \ldots, N-1$ をシャッフルした配列（長さ N): うまく言えない。。。
alpha	α	topics についてのハイパーパラメーター
beta	γ	words についてのハイパーパラメーター

4.9 Gibbs Sampling(1): ドキュメントのシャッフル

ドキュメントはシャッフルする。seq にシャッフルされたドキュメント番号が入っている。

4.10 Gibbs Sampling(2): ドキュメントループ

most outer loop の j のループで $n = seq[j]$ としてランダムの順にドキュメントを抽出している。またこの時ドキュメントに紐づいているデータを取り出す

```
n  = seq[j]
N  = len(Z[n])
Zn = Z[n]
Wn = W[n]
rands = npr.rand (N)
```

rands には N 個（ドキュメント長）の一様分布からサンプルした確率値が入っていて、これは後で更新される。

4.11 Gibbs Sampling(3): 単語位置ループ

ドキュメント内で単語を最初の単語から最後の単語まで走査する。

```
for i in range(N):
    Wni        = Wn[i]
    z          = Zn[i]
    NW[Wni,z] -= 1
    ND[n,z]   -= 1
    NWsum[z]  -= 1

    total = 0.0
```

この時、先頭から順に [1] で言うところの y_{il} と q_{il} を求めた後、関連する頻度行列 NW,ND,NWsum から当該アイテムのカウントをデクリメントして取り除いている。またこれから使う total 変数を初期化して 0.0 にしている。これにより $-i$ と書いてある c からの i の除外が行われる。

4.12 Gibbs Sampling(4): トピックループ 1

p[k] を求める。後の推測の章で詳しく説明するが、尤度の項を与えているかのような印象である。そして total でトピックについて周辺化している。

$$p(k) = \frac{C_{(v=y_{il}),k} + \beta}{C_k + |V|\beta} \{C_{ik} + \alpha\} \tag{1}$$

が本実装であるが (β は [1] では γ である)、[1] には

$$p(q_{i,l} = k | q_{-\ i,l}, y_i, \alpha, \gamma) \propto \frac{C^-_{v,k} + \gamma}{C^-_k + |V|\gamma} \frac{C^-_{i,k} + \alpha}{L_i + K\alpha} \tag{2}$$

とあり、少し次元が違う気がする。具体的には L_i に続く項がないため、大きな値になっている。また、次の記述があり

> Define c^-_{ivk} to be the same as c_{ivk} except it is compute by summing over all locations in document i except for q_{il}.[1]

q_{il} の分だけ -1 する必要があることが分かる。

4.13 Gibbs Sampling(5): トピックループ 2

$$X_l = P_l[0] \times P_l[1] \tag{3}$$

rands[i] は一様分布の乱数に足して次元を合わせるために total($p[k]$ の和) を掛けている。何回も掛けていくのかと思ったが、ループ内で一回だけである。これは後の p[k] の足し算の時に次元を合わせるためである。

```
            rands[i] = total * rands[i]
            total = 0.0
            znew  = 0
            for k in range(K):
                total += p[k]
                if rands[i] < total:
                    znew = k
                    break
```

p[k] を normalize する代わりにそのまま（次に生まれた）total に加算していく。加算値が乱数を超えたら記号（topic znew）の emit である。znew=0 で初期化されており、デフォルトでは topic0 が選ばれる。この時 total は

$$total_k = \sum_{k'=0}^{k} p[k'] \tag{4}$$

である。

あとは簡単で、znew を Z の配列に入れ直し、NW,ND,NWsum を更新しているだけである。

```
            Zn[i]         = znew
            NW[Wni,znew] += 1
            ND[n,znew]   += 1
            NWsum[znew]  += 1
```

このようにして変更された Z は、新しいトピック群を形成する。

5 perplexity の計算について

Gibbs Sampling の iteration を回す時、最初に perplexity を求める。この値は ldappl 関数で求まる。

```
def ldappl (W,Z,NW,ND,alpha,beta):
    L = datalen(W)
    lik = polyad (ND,alpha) + polyaw (NW,beta)
    return np.exp (- lik / L)

def datalen (W):
    return sum (map (lambda x: x.shape[0], W))
```

datalen 関数は W（ドキュメント毎の単語のリスト）の単語の個数をカウントアップする。これを L_i の総和として、L と置く。

文献 [1] より、perplexity は、

$$\mathrm{Perplexity}(p_{emp}, p) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\frac{1}{L_i}\sum_{l=1}^{L_i}\log q(y_{il})\right) \tag{5}$$

で求められることがわかる。ただ本実装は違う求め方をしている。

まず $L = \sum_{i=1}^{N} Li$ である。また N はドキュメント数である。

```
def gammaln (x):
    return scipy.special.gammaln (x)
```

ここでまず gammaln に注目する。$exp(gammaln(x)) = gammasgn(x) * gamma(x)$ だそうである。次にもう一度 perplexity の定義に戻る。

```
def ldappl (W,Z,NW,ND,alpha,beta):
    L = datalen(W)
    lik = polyad (ND,alpha) + polyaw (NW,beta)
    return np.exp (- lik / L)
```

lik（恐らく尤度）は polyad 関数と polyaw 関数の和である。polyad 関数と polyaw 関数を解析しよう。

```
def polyad (ND,alpha):
    D = ND.shape[0]
    K = ND.shape[1]
    nd = np.sum(ND,1)
    lik = np.sum (gammaln (K * alpha) - gammaln (K * alpha + nd))
    for n in xrange(D):
        lik += np.sum (gammaln (ND[n,:] + alpha) - gammaln (alpha))
    return lik
```

D はドキュメント数、K はトピック数である。([1] の場合 N と K である)。nd は row ごとに和を取っていて、row は D 個なので、各ドキュメントのトピックごとの頻度の和である。これは各ドキュメントごとの単語数に相当する。([1] の場合 $C_i = \sum_{k=1}^{K} C_{ik}$)。

$$lik = \log(\Gamma(K\alpha)) - \log(\Gamma([C_1 + K\alpha, C_2 + K\alpha, \ldots, C_N + K\alpha)) \tag{6}$$

またループ内は

$$lik = \sum_{i=1}^{N} \{\log(\Gamma([c_{i,0} + \alpha, c_{i,1} + \alpha, \ldots, c_{i,K} + \alpha])) - \log(\Gamma(\alpha))\} \tag{7}$$

同様な形式をしている polyaw も同様の式である。これらより perplexity が $\exp(-\frac{lik}{L})$ より求まる。

6 perplexity について推測の章

コードと [1] の両方から推測するに、次のような関係があるのではないだろうか

6.1 polyad について

```
def polyad (ND,alpha):
    D = ND.shape[0]
    K = ND.shape[1]
    nd = np.sum(ND,1)
    lik = np.sum (gammaln (K * alpha) - gammaln (K * alpha + nd))
    for n in xrange(D):
        lik += np.sum (gammaln (ND[n,:] + alpha) - gammaln (alpha))
    return lik
```

([1] の場合 $C_i = \sum_{k=1}^{K} C_{ik}$)。

$$lik = \log(\Gamma(K\alpha)) - \log(\Gamma([C_1 + K\alpha, C_2 + K\alpha, \ldots, C_N + K\alpha)) \tag{8}$$

またループ内は

$$lik = \sum_{i=1}^{N} \{\log(\Gamma([c_{i,0} + \alpha, c_{i,1} + \alpha, \ldots, c_{i,K} + \alpha])) - \log(\Gamma(\alpha))\} \tag{9}$$

ここで [1] を見てみると、

$$p(\mathbf{q}|\alpha) = \left(\frac{\Gamma(K\alpha)}{\Gamma(\alpha)^K}\right)^N \prod_{i=1}^{N} \frac{\prod_{k=1}^{K} \Gamma(C_{ik} + \alpha)}{\Gamma(L_i + K\alpha)} \tag{10}$$

(marginal prior だそうである)。
ここで C_i は L_i であることがわかるし、近い式であることがわかる。

6.2 polyaw について

コード中の beta は [1] での γ である。

```
def polyaw (NW,beta):
    V = NW.shape[0]
    K = NW.shape[1]
    nw = np.sum(NW,0)
    lik = np.sum (gammaln (V * beta) - gammaln (V * beta + nw))
    for k in xrange(K):
        lik += np.sum (gammaln (NW[:,k] + beta) - gammaln (beta))
    return lik
```

V は語彙数、K はトピック数である。nw は C_k となる。文献 [1] を引用する。

$$p(\mathbf{y}|\mathbf{q}, \gamma) = \left(\frac{\Gamma(V\gamma)}{\Gamma(\gamma)^V}\right)^K \prod_{k=1}^{K} \frac{\prod_{v=1}^{V} \Gamma(C_{vk} + \gamma)}{\Gamma(C_k + V\gamma)} \tag{11}$$

これもまた近い式であることがわかる。
gammaln を使っているため対数が掛かっているが、確率値の log である。
それを $\exp(-logprobability)$ として確率値に戻していると考えられる。

7 改善案

7.1 改善前

改善前のホットスポットはこんなコードである。

```
        total = 0.0
        for k in range(K):
            p[k] = (NW[Wni,k] + beta) / (NWsum[k] + V * beta) * \
                   (ND[n,k] + alpha)
            total += p[k]

        rands[i] = total * rands[i]
        total = 0.0
        znew  = 0
        for k in range(K):
            total += p[k]
            if rands[i] < total:
                znew = k
```

```
                break
```

8 改善後

次の改善を提案する。大きな効果はなかったが、次元が合うなどの作用がある。

```
    for i in range(N):
        Wni        = Wn[i]
        z          = Zn[i]
        NW[Wni,z] -= 1
        ND[n,z]   -= 1
        NWsum[z]  -= 1
        doc_len = N - 1

        normalizer = 0.0
        for k in range(K):
            p[k] = (ND[n,k] + beta) / (doc_len + beta * K) * (NW[Wni,k] + alpha) / (NWsum[k]
                + alpha * V)
            normalizer += p[k]

        total = 0.0
        znew  = 0
        for k in range(K):
            total += p[k] / normalizer
            if rands[i] < total:
                znew = k
                break
```

doc_len として新しい項が $p[k]$ に追加されていると同時に、total でも止める試算値を normalize で割っている。おそらく p[k] が若干正確になっている。

9 テストセット用に構築したコーパス環境とツール

ツール名	Python のバージョン	付記
lda.py	python2	持橋先生ツール
preNIPS.py	python2	NIPS コーパス→ svm
preNewsGroups.py	python2	newsgroup20 → svm
preJaText8.py	python3	ja.text8 → svm
visLDA.py	python3	.wz ファイル→結果 csv
coolcutter.py	python3	.svm ファイルシュリンクツール
scikit-LDA.py	python3	scikit-learn で LDA
ExtractCorpus.py	python3	ja.text8(new) 作成ツール

またこれらのツールのうち木村の自作のツールのほとんどは.words ファイルを副次的に扱う。

9.1 使い方、コマンドラインオプションなど

9.1.1 preNIPS.py

usage は次の通り。

```
MacBook-Pro:nlp kimrin$ python preNIPS.py
usage: preNIPS.py NIPS_1987-2015.csv wordsfile.txt
       output sparse matrix file that work for LDA.py
       also put words list file.
MacBook-Pro:nlp kimrin$
```

NIPS コーパス自体は一つの大きな csv ファイルである。これは row が単語、column が document となっている。これを転地し、svm の疎行列もどきに直す。引数として NIPS コーパスの csv ファイルと出力される words ファイルを指定する。これから lda するときは NIPS という prefix の付いたファイル群を生成するので、このとき NIPS.words を words ファイルとする。注意して欲しいのは words ファイルは 1column 目をそのまま切り出したものなので、alphabetical order になっている点である。他のコーパスは頻度順である。

```
MacBook-Pro:nlp kimrin$ python preNIPS.py ../data/NIPS_1987-2015.csv ../data/NIPS.words
reading 0...
reading 100...
...
reading 11300...
reading 11400...
writing 0...
writing 100...
writing 200...
...
writing 5700...
writing 5800...
generated ../data/NIPS_1987-2015.csv.pseudo.svm and ../data/NIPS.words
next, run LDA.py to obtain latent dirichlet allocations.
```

../data/NIPS_1987-2015.csv.pseudo.svm ファイルが出来上がる。同時に../data/NIPS.words ファイルもできあがる。

9.1.2 preNewsGroups.py

usage は次の通りである。

```
MacBook-Pro:nlp kimrin$ python preNewsGroups.py
Usage:
  preNewsGroups.py [--pt] <news_groups_dir> <pseudo_svm_file> <prefix>
  preNewsGroups.py (-e | --each) <a_file>
  preNewsGroups.py (-h | --help)
  preNewsGroups.py --version
MacBook-Pro:nlp kimrin$
```

20newsgroups コーパスは一つのドキュメントが一つのファイルに入っている（NetNews の記事）。これを recursive に集めてきて、svm ファイルを作る。

```
MacBook-Pro:nlp kimrin$ python preNewsGroups.py ../data/20news-bydate/ 20news.svm 20news
../data/20news-bydate/20news-bydate-test/talk.politics.mideast
```

```
...
../data/20news-bydate/20news-bydate-train/talk.religion.misc
MacBook-Pro:nlp kimrin$
```

20news.svm ファイルに pseudo svm ファイルが、20news.words に words ファイルが格納される。test と train が別れているが両方コーパスとして扱う。words ファイルのオーダーは頻度オーダーである。

9.1.3 preJaText8.py

ja.text8（改良版）コーパスを読み込んで pseudo svm ファイルと words ファイルを出力する。usage は次の通り。

```
MacBook-Pro:nlp kimrin$ python3 preJaText8.py
Usage:
  preJaText8.py <corpus_file> <prefix>
  preJaText8.py (-h | --help)
  preJaText8.py --version
MacBook-Pro:nlp kimrin$
```

python3 ファイルである。コーパスファイルと prefix を与える。ja.text8.svm と ja.text8.words が出来上がる。

```
MacBook-Pro:nlp kimrin$ python3 preJaText8.py ../ja.text8/ja.text8.20180601.100MB ja.text8
36435 articles are processed.
MacBook-Pro:nlp kimrin$
```

9.1.4 lda.py

（持橋先生の作成されたシステムである）。LDA を行い結果を.wz ファイルに格納する。.wz ファイルは単語番号とトピック番号の対が、コーパス内の単語の数だけ格納されたテキストファイルである。そのほかハイパーパラメータなども保存される。

```
MacBook-Pro:nlp kimrin$ python python/lda.py
usage: lda.py OPTIONS train model
OPTIONS
 -K topics number of topics in LDA
 -N iters  number of Gibbs iterations
 -a alpha  Dirichlet hyperparameter on topics
 -b beta   Dirichlet hyperparameter on words
 -h        displays this help
$Id: lda.py,v 1.10 2016/02/06 14:05:38 daichi Exp $
MacBook-Pro:nlp kimrin$
```

train と書いてあるところに.svm ファイルを指定する。model の代わりに prefix を指定する。このとき prefix.words があらかじめ作られているのなら、prefix をそれに合わせると都合が良い。

```
MacBook-Pro:nlp kimrin$ python python/lda.py -K 10 -N 100 --alpha=0.01 ../data/ja.text8.svm
    ../data/ja.text8
LDA: K = 10, iters = 100, alpha = 0.01, beta = 0.01
loading data.. documents = 35938, lexicon = 17416, nwords = 8254429
initializing..
100,4384.1300
saving model..
```

```
done.
MacBook-Pro:nlp kimrin$
```

log を csv として読み込むためのパッチが当ててあるので結果が少し違うが、iteration の数と perplexity が出るようになっている。

9.1.5 visLDA.py

lda.py の結果をわかりやすい csv ファイルの形にする整形ツールである。python3 ファイルである。頻度の高い単語の順に、各コラムにトピックを割り当てる。カッコ内の数字はそのトピック内での頻度である。いくつか例外があり、= はマクロとなってしまうため、コーパス内に = があってこれが結果に反映される場合はマクロを無効化して差し支えない。また cp932（windows の shift-jis）にない文字が含まれている場合、これを別の文字で置き換えるが稀である。usage は次の通りである。

```
MacBook-Pro:nlp kimrin$ python3 visLDA.py
visLDA.py: visualize allocations of LDA.
But this time, only emits one csv file(columns = topics, rows = word rank).

 $ python visLDA.py prefix words.txt
   prefix: corpus name (such as model, NIPS).
   words.txt: word file (for example, NIPS.words)
MacBook-Pro:nlp kimrin$
```

まず.wz ファイルが含まれている prefix を指定する。同時に.words ファイルのファイル名を指定する。本当は prefix だけにしてシンプルなコマンドラインにしたかったができなかった。出力として、prefix.topics.csv が出力される。いつも Excel で開いているが Numbers でも開けるかもしれない。K の数だけカラムができる。

```
MacBook-Pro:nlp kimrin$ python3 visLDA.py ../data/ja.text8 ../data/ja.text8.words
make bag of words: doc id 100
make bag of words: doc id 200
...
make bag of words: doc id 35900
read 35938 documents. 10 topics.
draw 9366 records.
MacBook-Pro:nlp kimrin$
```

9.1.6 coolcutter.py

後述の scikit-LDA.py の前処理に感化されて作った素性削除ツールである。.svm ファイルを入力し、.cool.svm ファイルを出力する。.cool.svm ファイルが素性削除されてスリムになったファイルである。あくまで客観的に言えないが、これを通すと結果がある程度妥当になる。ただ scikit-LDA.py のところでも述べるが、perplexity は scikit-LDA.py の方がかなり低い。

```
MacBook-Pro:nlp kimrin$ python3 coolcutter.py
Usage:
  coolcutter.py <prefix>
  coolcutter.py (-h | --help)
  coolcutter.py --version
```

.svm ファイルの prefix をコマンドラインに指定する。

```
MacBook-Pro:nlp kimrin$ python3 coolcutter.py ../data/ja.text8
```

```
1. calculated df.
2. omit exceed max df and lower df in probability...
3. calculate tf in corpus.
4. sort it
5. perform deletion
write ../data/ja.text8.cool.svm.
```

```
MacBook-Pro:nlp kimrin$ ls -l ../data/ja.text8.svm ../data/ja.text8.cool.svm
-rw-r--r-- 1 kimrin staff   303286 Jun 25 20:40 ../data/ja.text8.cool.svm
-rw-r--r-- 1 kimrin staff 28795790 Jun 14 23:34 ../data/ja.text8.svm
MacBook-Pro:nlp kimrin$
```

1/100 くらいになるまでごっそり素性を削除する。perplexity が改善するが、scikit-learn の LDA には及ばない。

9.1.7 scikit-LDA.py

scikit-learn に LDA があると聞いてお試しで解析してみた。あくまでお試しなので ja.text8 コーパスがハードコーディングされている。環境に合わせて変更するか、もしもっと使いたければコマンドラインを拡張して欲しい。NMF モデルと LDA モデルの両方が試される。LDA は tf しか使わない。入力としてはスペース区切り、一行 1 document の形式で十分である。

前処理が施されている。df が 0.95 以上のものは取り除く（確率値として）。df が 2 以下のものは取り除く（頻度として）。素性は 1000 で。

結果も併せて掲載する。

```
MacBook-Pro:nlp kimrin$ python3 scikit-LDA.py
Loading dataset...
done in 1.165s.
Extracting tf-idf features for NMF...
done in 8.229s.
Extracting tf features for LDA...
done in 8.601s.

Fitting the NMF model (Frobenius norm) with tf-idf features, n_samples=36436 and n_features=1000...
done in 4.889s.

Topics in NMF model (Frobenius norm):
Topic #0: する こと いる れる ない ある など よう もの ため この から として その という なる 場合 られる また によって
Topic #1: 選手 出場 リーグ 試合 シーズン チーム 大会 サッカー 優勝 移籍 選手権 代表 出身 プロ fc 野球 所属 オリンピック 契約 クラブ
Topic #2: アルバム 発売 シングル リリース 収録 作曲 バンド 楽曲 cd the 音楽 いる オリジナル ライブ レコード から ベスト 発表 歌手 限定
Topic #3: 日本 教授 大学 研究 東京 学校 卒業 高等 出身 博士 名誉 学者 大学院 文学 教育 所属 専門 生まれなど 活動
Topic #4: 放送 番組 テレビ ラジオ 00 から 出演 時間 まで ドラマ nhk 制作 系列 10 30 開始 終了 コーナー情報 担当
```

Topic #5: ある いる ホーム 鉄道 位置 国道 あり 道路 km 路線 どう バス 現在 一般 から 列車 jr 設置 結ぶする
Topic #6: 小学校 市立 中学校 学校 丁目 高等 大阪 ある こう 平成 教育 する 以下 県立 通り 統合 現在 神戸福岡 住宅
Topic #7: 映画 作品 監督 出演 公開 女優 俳優 製作 主演 ドラマ 日本 テレビ 漫画 受賞 シリーズ デビュー 小説 撮影 出身 いる
Topic #8: から あっ なっ として 時代 現在 られ 昭和 明治 10 まで 12 いる 11 この ため 江戸 なり その により
Topic #9: 人口 イタリア 共和 自治体 基礎 以下 km 通り ある 一つ する 位置 都市 距離 面積 行政 地域 から示す 2000

Fitting the NMF model (generalized Kullback-Leibler divergence) with tf-idf features, n_samples=3643
done in 74.608s.

Topics in NMF model (generalized Kullback-Leibler divergence):
Topic #0: する ある いる れる こと として など また あり ない から なる ため なっ もの この よう その られ によって
Topic #1: 出身 日本 所属 として 活動 から 卒業 東京 12 11 デビュー 同年 高校 務め 代表 その後 参加 10 大学 より
Topic #2: 発売 から 放送 番組 いる シングル アルバム 音楽 収録 まで リリース テレビ 作曲 制作 楽曲 より開始 時間 作品 ラジオ
Topic #3: 日本 学校 東京 高等 教授 研究 など 大学 株式会社 設立 教育 生まれ 専門 現在 卒業 本社 ある 法人 会社 運営
Topic #4: 選手 する 出場 開催 大会 優勝 選手権 チーム 試合 リーグ サッカー 行わ シーズン プロ 獲得 野球から 代表 競技 世界
Topic #5: ある 位置 いる 現在 鉄道 から 市立 小学校 一般 以下 どう 道路 存在 地域 結ぶ 都市 通り 路線 設置 番号
Topic #6: する こと ため その なっ なる この れる られ しかし ない として という よう あっ なかっ これだっ なり また
Topic #7: 日本 映画 作品 ある 監督 いる する 漫画 など アメリカ合衆国 舞台 俳優 英語 連載 女性 公開 小説 男性 出演 による
Topic #8: あっ 現在 時代 として なっ から まで いる 12 10 11 により 昭和 られ 明治 なり 江戸 ため 当時 15
Topic #9: ある 生まれ イタリア 共和 ドイツ 人口 フランス 一つ から 発見 イギリス 基礎 政治 通り アメリカ 世界 自治体 世紀 知ら 以下

Fitting LDA models with tf features, n_samples=36436 and n_features=1000...
iteration: 1 of max_iter: 5, perplexity: 369.1543
iteration: 2 of max_iter: 5, perplexity: 364.6237
iteration: 3 of max_iter: 5, perplexity: 363.0649
iteration: 4 of max_iter: 5, perplexity: 362.1714
iteration: 5 of max_iter: 5, perplexity: 361.5199
done in 147.643s.

```
Topics in LDA model:
Topic #0: 研究 大学 学校 教授 教育 科学 小惑星 高等 文化 学者 社会 博士 日本 植物 技術 専門 研究所 建築病院 経済
Topic #1: ある いる から する なっ 鉄道 昭和 あっ 平成 まで 現在 設置 として 列車 車両 路線 バス 地区 10 建設
Topic #2: 放送 番組 から 音楽 いる アルバム 出演 テレビ ある として 10 発売 まで シングル 作曲 収録 活動 12 11 ドラマ
Topic #3: 日本 など 映画 いる として 東京 ある 会社 する から 出身 活動 昭和 監督 卒業 設立 株式会社 事業 企業 現在
Topic #4: する ある こと ない について れる 場合 において よう もの その なる という として いる 問題 なら できる 市立 小学校
Topic #5: から する ドイツ なっ として あっ フランス この 政府 ため イギリス 戦争 アメリカ まで 海軍 ロシア こと 主義 部隊 10
Topic #6: から 選手 チーム 試合 出場 大会 シーズン リーグ 優勝 として 代表 なっ 選手権 する 野球 プロ 10 開催 世界 獲得
Topic #7: から こと いる する なっ この ある あっ その として ため という られ ない よう れる だっ 時代なかっ なる
Topic #8: ある いる する れる こと など から この として ない もの ため あり よう られ その また によって られる なる
Topic #9: いる する ある こと から なっ 作品 として など ない ゲーム シリーズ 開発 登場 使用 ため 発売れる また より

MacBook-Pro:nlp kimrin$
```

9.1.8 ExtractCorpus.py

これは今回の LDA 用前処理としてはちょっと特殊な部類に入るのであるが、コーパス作成のためのスクリプトである。

ja.text8 を fork し、一行 1 ドキュメントになるように修正した。また ja.text8 が使っていた元の wikipedia のファイルは wikipedia サーバーになかったので最新の同様のファイルを使って作成した。このスクリプトを実行してコーパスを作るのではなく、このリポジトリの setup.sh を実行すると良い塩梅にこのスクリプトが呼ばれる。

基本的には MeCab で分かち書きし、一つのドキュメントを一つの行に収める。100MB に達しそうなところで clamp する。

```
MacBook-Pro:nlp kimrin$ ls -l ../ja.text8/ja.text8.20180601.100MB
-rw-r--r-- 1 kimrin staff 100036067 Jun 14 11:42 ../ja.text8/ja.text8.20180601.100MB
MacBook-Pro:nlp kimrin$
```

参考文献

[1] Kevin P. Murphy: Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series) The MIT Press, 2012.

[2] Blei, David M. and Ng, Andrew Y. and Jordan, Michael I.: Latent Dirichlet Allocation, J. Mach. Learn. Res. 3/1, volume 3, 2003.