3　単語の統計モデル

3.1　文字から単語へ

2 章では, 文字の統計モデルを題材に, 確率の基本と文字 n グラム言語モデル, およびモデルの評価方法について学んできました.

ここまでは簡単のために文字だけのモデルを考えてきましたが, 実際には言語には普通は「単語」の概念があり, 単語を使った方が, よりよいテキストのモデルになると考えられます. たとえば, 英語で “tall” の次に続く言葉は “tree” や “boy” であり, “reason” が来ることはまずないことは, 単語としてこの言葉の意味を考えず, “-ll” で終わる文字列として見ていたのでは難しいでしょう.

なお, 「単語」という概念があっても, それが現在の英語のように, 空白文字で分かち書きされているとは限りません. 日本語や中国語, タイ語のような東アジアの言語には空白はありませんし, さらにはラテン語や古典ギリシャ語, 古くは英語も, もともとは図 3.1 のように単語間に空白を開けず, 続け書きされていました. これは, *scriptio continua* (ラテン語で「続け書き」の意味) とよばれています.

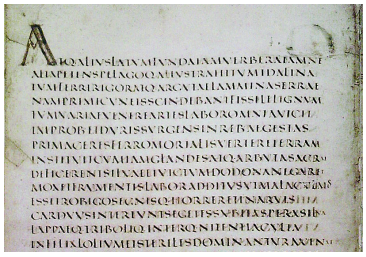


図 3.1: ヴェルギリウスのラテン語のテキスト (西暦 141 年ごろ) での, 単語の続け書きの例. *1

一方で, “New York” のような地名, “with respect to” のような慣用句はそれぞれ “N.Y.”, “w.r.t.” とも書かれることからわかるように, 空白があっても, 実質的に単語の区切りであるとは限りません. また, “other than” のよ

*1　https://en.wikipedia.org/wiki/Scriptio_continua より引用, 一部.

うなすべての慣用句を単語として認めるかが決まっているわけではありません. よって, 何が「単語」であるかという基準には本質的に曖昧性があり, 唯一の正解があるわけではありませんが,[*2] 多くの言語は現在では空白で区切って書かれますし, 日本語や中国語も, 最適とはいえなくても, 下に述べるように標準的な方法で単語に区切ることができますので, 本書でもそれを使っていくことにします. 上のような慣用句を統計的に自動認識する方法については, この後の 3.2.2 節を参照してください.

文字列を単語に分解することは**単語分割**, あるいは動詞や名詞といった品詞の付与や活用形の認識も行う場合は**形態素解析**とよばれています. たとえば, 標準的な形態素解析器の一つである MeCab[*3] を使えば, 日本語の文字列を簡単に単語に区切ることができます. Python の場合, これにはまず, `MeCab` モジュールをインストールする必要があります. 詳細は付録**??**に譲りますが, Google Colaboratory の Python 環境を使っている場合は, 執筆時では次のようにすればインストールすることができます (>は Colaboratory のプロンプト).

```
> !pip install mecab-python3
> !pip install unidic-lite
```

この上で, 下のようにすれば文字列を単語に分割することができます.

```
import MeCab
tagger = MeCab.Tagger ("-Owakati")
s = "これは日本語の文字列です。"
tagger.parse(s).split()  # 結果を空白文字で分割する
⇒ ['これ', 'は', '日本語', 'の', '文字', '列', 'です', '。']
```

したがって, テキスト全体は次のようにして単語に分解することができます.

```
with open ('ginga.txt', 'r') as fh:
    for line in fh:
        buf = line.rstrip('\n')  # 行末の改行文字を削除
        if len(buf) > 0:         # 空行でない場合
```

*2 筆者は, この問題に対して「観測文字列の確率を最大化する分割が単語である」という立場で教師なし形態素解析[46]の研究を行い, 情報理論的な解を与えています. ただし, これは確率モデルに依存しますので, 人間の持っている「単語」という概念にかなり近くなるものの, 必ず一致するわけではありません.

*3 MeCab は奈良先端大 (当時) の工藤拓氏によって開発された, 条件付き確率場 (CRF) に基づく形態素解析器で, `https://taku910.github.io/mecab/`で配布されています.

```
            words = tagger.parse(buf).split()
            print (words)
⇒ [’銀河’, ’鉄道’, ’の’, ’夜’]
  [’一’, ’、’, ’午后’, ’の’, ’授業’]
  [’「’, ’では’, ’みなさん’, ’は’, ’、’, ’そういう’, ’ふう’, ’
  に’, ’川’, ’だ’, ’と’, ’云わ’, ’れ’, ’たり’, ...
```

このまま解析に使うことも可能ですが, 後の処理のために次のように, 単語に分割したテキストを別ファイルに保存しておくとよいでしょう. 下のようにすると, `ginga.split.txt` に単語分割されたテキストが保存されます.

```
with open (’ginga.split.txt’, ’w’) as oh:  # 出力ファイル
    with open (’ginga.txt’, ’r’) as fh:    # 入力ファイル
        for line in fh:
            buf = line.rstrip(’\n’)
            if len(buf) > 0:               # 空行は無視する
                words = tagger.parse(buf).split()
                oh.write (’ ’.join(words) + ’\n’)
```

なお, 形態素解析器にはほかにも kuromoji, GiNZA, KyTea, …など様々なものがあります [*4] ので, 使いやすいものを用いるとよいでしょう.

3.2　単語の統計と巾乗則

さて, こうしてテキストが単語に分けられたとき, 文字のモデルと最も異なっている点は何でしょうか. 最も重要なのは, 「単語の種類は無限にある」ということでしょう. それぞれの言語で使われる文字の種類は有限ですが, 文字の組み合わせである単語は, 長さに通常は制限がありませんので, いくらでも多くの単語を作り出すことができます. [*5]

---

*4　これらの違いは, 提供している機能の違いもさることながら, 最も大きいのは「単語」をどう定義するかという違いです. たとえば奈良先端大で開発された MeCab などで標準的に使われている IPA 辞書では IPA 品詞体系が採用されており, いっぽう京大で開発された **JUMAN** では, 益岡・田窪文法をもとにした別の JUMAN 品詞体系が採用されています. 単語分割が異なるとモデルも違ってきてしまいますので, モデルを学習する際と, それを使用して新しいテキストを解析する際には一般に, 形態素解析器を揃える必要があります.

*5　日本語では「リュウグウノオトヒメノモトユイノキリハズシ」のような植物名, 英語では『メアリー・ポピンズ』に現れる “supercalifragilisticexpialidocious” のような言葉が有名ですが, もっと長いものも存在し, これらは**長大語**とよばれています.

とはいえ, 計算機上で無限の語彙を表すのは難しいため [*6], ほとんどの場合は, 一定の基準で語彙を選ぶことになります. 実際に, われわれが普段用いている辞書はこうして一定の語彙を選んだものです. 表 3.1 に, 日本語および英語の主な辞書に掲載されている語彙 (見出し語の数) の例をまとめました. これを見ると, 単語として必要な語彙の種類は文字の種類よりはるかに大きく, 最低でも数万語, 多い場合には数十万語を超えることがわかります. つまり, 統計モデルとしてみると, 単語を考える統計モデルは出力が数万次元を超える**超高次元の統計モデル**になるということです. これが, 文字の統計モデルとの大きな違いです.

それでは, 実際のテキストにはどれくらいの語彙が含まれているのでしょうか. 3.1 節で示した方法でテキストが空白で単語に分かれているとき, 次のようにすれば単語の総数と, それぞれの単語の頻度を数えることができます.

```
from collections import defaultdict
freq = defaultdict (int)
with open ('ginga.split.txt', 'r') as fh:
    for line in fh:
        words = line.rstrip('\n').split()
        for word in words:
            freq[word] += 1

for w,c in freq.items():
    print ('%s -> %d' % (w,c))
```

```
⇒ 銀河 -> 25
  鉄道 -> 4
  の -> 1266
```

表 3.1: 日本語および英語の標準的な辞書に含まれる語彙 (見出し語) の大きさ.

言語	辞書	語彙 (約)
日本語	岩波国語辞典 第八版	67,000 語
	広辞苑 第七版	250,000 語
英語	Longman LDCOE	45,000 語
	リーダーズ英和辞典	280,000 語
	Oxford English Dictionary	600,000 語

*6 筆者による教師なし形態素解析のための言語モデル NPYLM [20]は, 文字 ∞ グラムと単語 n グラムを組み合わせることで, 無限の語彙を扱うことが可能です. ただし, これは本書のレベルを大きく超えるため, 説明は割愛しています.

```
    夜 -> 6
    一 -> 45
    、 -> 988
    午后 -> 2
    授業 -> 2
    「 -> 293
    では -> 10
    みなさん -> 4
    ...

    len(freq)
⇒ 2594
```

「銀河鉄道の夜」では語彙の大きさは 2594 であり, 各単語の頻度は上のようになっていることがわかります. 他のテキストについても同様に数えた語彙数を, 表 3.2 に示しました. 「銀河鉄道の夜」や「不思議の国のアリス」といった (計算機にとっては) 短いテキストでは語彙は数千語程度ですが, 一般的な中規模のテキストでは数万語以上, 大きなテキストでは数十万語や数百万語を超えることがわかります. *7 なお, このように一般的に共有されているテキストデータおよび (あれば) 付随データのことを, **コーパス**ともいいます.

コーパスにはさまざまなものがありますが, 1967 年に編纂された Brown コーパス[49]は分野が偏らないようにサンプリングされた (これを**均衡コーパス**といいます), 最初の大きなコーパスです. アメリカ英語約 100 万語のテキストとその品詞からなっており, 現在では NLTK のようなツールキットにも付属して公開されています *8. 現代の均衡コーパスとしては, 英国では 1 億語の British National Corpus (BNC)*9, 米国では 1500 万語の American National Corpus (ANC)*10 があり, いずれもデータをフリーでダウンロードできます. 日本語では, 国立国語研究所が約 1 億語の現代日本語書き言葉均衡コーパス (BCCWJ*11) を公開しており, オンライン検索は無料で, オフラインのデータは有料で使用す

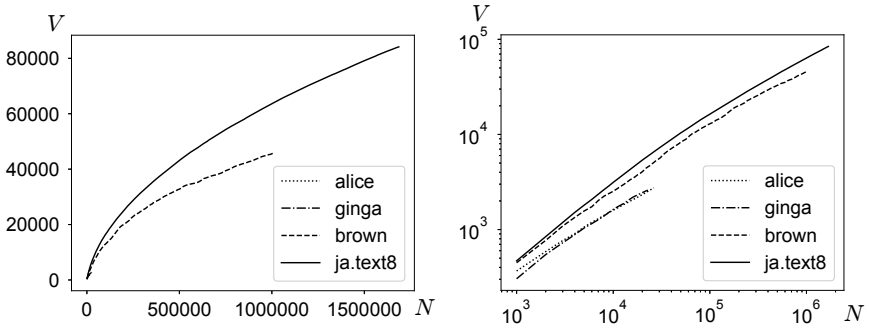
*7　Web から取得した 1 兆語の英語テキストから求めた Google 1T 5-gram データ[47]では語彙は 1350 万語, 日本語 2500 億語の 7-gram データ[48]では 256 万語にものぼります.

*8　http://www.nltk.org/nltk_data/

*9　http://www.natcorp.ox.ac.uk/

*10　https://anc.org/

*11　Balanced Corpus of Contemporary Written Japanese の略. https://clrd.ninjal.ac.jp/bccwj/ から使用・入手することができます.



(a) 線形スケールの場合 (b) 両対数スケールの場合

図 3.2: テキストの長さ N と語彙の大きさ V の関係 (Heaps の法則). `alice`:『不思議の国のアリス』, `ginga`:『銀河鉄道の夜』, `brown`: Brown コーパス, `ja.text8`:日本語 text8 コーパスをそれぞれ表しています. `alice`, `ginga` は短いため, 線形スケールではほとんど見えなくなっています.

ることができます. また有料ではありますが, 毎日新聞, 朝日新聞, 日経新聞といった新聞のテキストも近年のものはすべて電子化されており [*12], 購入すればコーパスとして使用することができます. 多言語のコーパスとしては, ドイツで公開されている Leipzig コーパス[50]には 100 言語以上の文が, それぞれ 1 万文から 100 万文含まれており (日本語も Web とニュースの文がそれぞれ 100 万文あります), フリーでダウンロードすることができます [*13].

また, 3.5 節で説明する単語ベクトルの学習などを手軽に試すためのコーパスとして, Wikipedia からランダムに 100MB 分のテキストを抽出した text8[*14], および日本語版の ja.text8[*15] はフリーのため, 研究や開発目的で広く使われています. サポートページに, Brown コーパスのテキストを `brown.txt`, text8 および日本語 text8 を `text8`, `ja.text8` として置いておきました. 元々の text8 には改行がないため, 改行を復元したものはそれぞれ `text8.txt`, `ja.text8.txt` となっています.

*12 https://www.nichigai.co.jp/sales/corpus.html
*13 https://wortschatz.uni-leipzig.de/en/download
*14 http://mattmahoney.net/dc/textdata
*15 https://github.com/Hironsan/ja.text8

3.2.1　Heaps の法則

単語の種類に制限はありませんから, 語彙は一般に, テキストが長くなるほど増えていきます. 以下では, テキストの長さは単語数で数えることにしましょう. 上の実行例で, テキストを読みながら頻度辞書 `freq` の大きさ `len(freq)` を見れば, 語彙がどのように増えるのかを確かめることができます. 図 3.2 に, こうして調べたテキストの長さ N と語彙 V の関係を示しました. テキストが長くなるほど, 語彙は増えてきれいな曲線を描き, 対数スケールで見るとテキストの言語やジャンルによって多少の違いはあるものの, おおむね同じように語彙が直線的に伸びていくことがわかります. このことは, **Heaps の法則**とよばれています.[*16] Heaps の法則は, 次の式で表されます.

$$V = K \cdot N^{\gamma} \qquad (\textbf{Heaps の法則}) \tag{3.1}$$

ここで V はテキストを長さ N まで見たときに含まれる語彙の大きさ, γ と K はコーパスに依存する定数です. 一般的な英語の場合は γ は 0.5 前後, K は 10 から 100 前後の値になることが知られています. 式(3.1)は, 両辺の対数をとれば

$$\log V = \log K + \gamma \log N \tag{3.2}$$

になりますから, $\log V$ と $\log N$ は傾き γ の比例関係となります. これが, 両対数スケールの図 3.2(b) で直線関係が現れている理由です.

表 3.2: テキストに含まれる語彙の大きさの例. 頻度を一定以上に限った場合についても同時に示しました.

テキスト	長さ	語彙	頻度 ≥ 2	頻度 ≥ 10
`alice.txt`	26396	2748	1471	379
`brown.txt`	1012603	46055	26481	8395
日本語 text8	15160499	249629	126690	45024
毎日新聞 2011 年度	21805730	129971	88669	42465
New York Times 2008 年度	65333240	249610	158735	72554

*16　Heaps の法則の名前は, 情報検索の分野で Harold Stanley Heaps が 1978 年に出版した本[51]によるものですが, これより前に, 初期の計量言語学者である Gustav Herdan (1897–1968) が 1960 年にこの法則を発見していました[52]. よって, これを Herdan-Heaps の法則ということもあります.

なお, 次節で説明する Zipf の法則を認めれば, Heaps の法則は Zipf の法則から導くことができます. この後で説明するように, 頻度順に単語を並べたとき, 単語の頻度 f が順位 r の逆数に比例する式(3.10), すなわち

$$f \propto r^{-\alpha} \tag{3.3}$$

が Zipf の法則ですから, 確率分布にするための正規化定数は

$$Z = \sum_{r=1}^{\infty} r^{-\alpha} = 1 + \frac{1}{2^\alpha} + \frac{1}{3^\alpha} + \frac{1}{4^\alpha} + \cdots \tag{3.4}$$

です. この Z は $\alpha > 1$ のとき, 一定の値に収束することが知られています.[*17] よって Zipf の法則の下では, 頻度順で r 番目の単語の確率は

$$p(r) = \frac{1}{Z} r^{-\alpha} \tag{3.5}$$

と表されるわけです.

いま, テキストを $N-1$ 語読んだときに, 全部で $V-1$ 語の語彙が出現したとしましょう. ここで次に読んだ N 語目の単語が語彙にない, 新しい単語だったとすると語彙は V 個に増え, この単語の頻度は 1 で, 確率は $1/N$ になります. この単語は頻度順では V 番目ですから, 式(3.5)から

$$\frac{1}{Z} V^{-\alpha} = \frac{1}{N} \tag{3.6}$$

が成り立つはずです. 両辺の対数をとって整理すれば,

$$\begin{aligned} -\alpha \log V &= \log Z - \log N \\ \therefore \quad V &= Z^{-1/\alpha} \cdot N^{1/\alpha} \end{aligned} \tag{3.7}$$

となり, Heaps の法則が $K = Z^{-1/\alpha}, \gamma = 1/\alpha$ を係数として得られます. 一般に言語では α は正確には 1 よりやや大きく, $\alpha = 1 \sim 2$ 程度ですから, γ は 0.5 程度になり, 実際の観察ともよく合致しています.

---

*17 数学ではリーマンのゼータ関数 $\zeta(\alpha)$ と呼ばれています. この場合, α に依存する定数 $\phi(\alpha) < 0.5772$ (右辺はオイラーの定数) を使って, $Z = 1/(\alpha-1) + \phi(\alpha)$ と表すことができます.

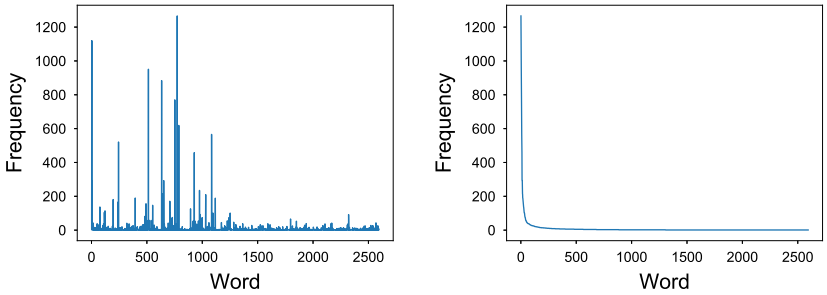
3.2.2　Zipf の法則

上ではテキストに現れる単語を数えて語彙を求めましたが, これらの単語がすべて, 同じ頻度で出現するわけではありません. そこで, 2.1 節で文字について行ったように, 単語を出現頻度で並べて表示してみましょう. これは文字の場合とまったく同様に, 次のようにして行うことができます.

```
    for w,c in sorted (freq.items(),
                       key=lambda x: x[1], reverse=True):
        print ('%s -> %d' % (w,c))
⇒ の -> 1266
  。 -> 1120
  、 -> 988
  た -> 951
  て -> 884
  に -> 770
  は -> 619
  を -> 566
  ...
  鷺 -> 19
  気 -> 18
  光っ -> 18
  そっち -> 18
  ...
  函 -> 1
  橄欖 -> 1
  堅く -> 1
  握っ -> 1
  便り -> 1
  放課後 -> 1
  知らせよ -> 1
```

この結果からわかる通り, 句読点を除くと「の」「に」「が」といった一部の単語に頻度が集中しており, 一方でテキストに一度しか出現していないような「函」「橄欖」といった言葉が大量にあることがわかります.[*18] これらは 1 回し

*18　こうした一度しか出現しなかった語のことを, コーパス言語学では孤語あるいは hapax legomenon (ギリシャ語で「一度だけ書かれた」の意味) といいます. これは, あくまで与えられたテキストについての概念であることに注意してください. 例えば, 「放課後」はこのテキストでは孤語ですが, 一般にはごくありふれた単語です.



(a) 単語を横軸に辞書順に並べた場合. (b) 単語を横軸に頻度順に並べた場合.

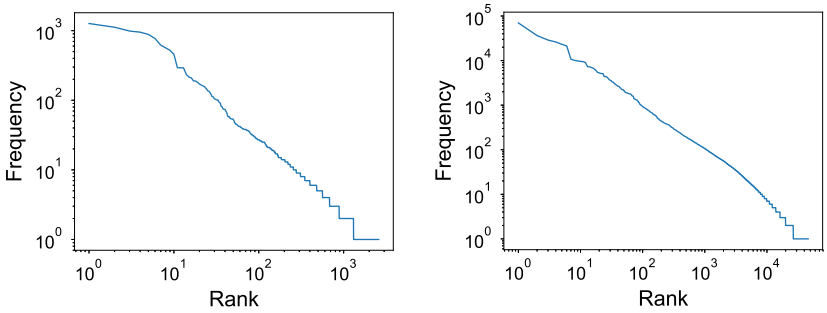
図 3.3: 『銀河鉄道の夜』における単語の頻度.

か出現していないために情報が少なく, 統計的な分析からは除いた方がよいでしょう. 一般に, テキスト全体を通した頻度がたとえば 10 以上 (大きいテキストならば 100 以上) のように, 基準を決めて語彙を選択することがよく行われます. 表 3.2 の最後の列に, 頻度 10 以上の単語を用いた場合の語彙の大きさを示しました. ただし今は, そうした語彙の選択は行わないことにします.

それでは, 上で求めた単語の頻度は, 全体としてはどのように分布しているのでしょうか. 図 3.3(a) に, 『銀河鉄道の夜』に現れる 2594 種類の単語を横軸に辞書順にとり, 縦軸にその頻度をとったプロットを示しました. 先にみたように, 日本語では「の」「に」「が」といった一部の単語が高い頻度を持っていますので, ヒストグラムにところどころ, 突出した値があることがわかります. 横軸の単語を頻度順で並び換えると, 図 3.3(b) のようになります. 確かに, 非常に少数の単語に頻度が集中しているのを見てとることができますね.

ただ, 頻度があまりに一部の単語に集中しているため, 様子を詳しく見るために, 縦軸を頻度 f の対数すなわち $\log f$ で, 横軸も同様に頻度の順位 r の対数 $\log r$ で両対数プロットにしてみましょう. こうすると, 図 3.3(b) は図 3.4(a) のように表されます. この図をよく見ると, $\log f$ と $\log r$ の間には傾きがほぼ -1の比例関係があることがわかります. すなわち, ほぼ

$$\log f = -\log r + b \qquad (b \text{ は定数}) \tag{3.8}$$



(a) 『銀河鉄道の夜』の場合．図 3.3(b) を両軸について対数でプロットしたもの．

(b) Brown コーパスで同様に計算した場合．

図 3.4: 単語の頻度順位と出現頻度の両対数プロット．順位と頻度が反比例する Zipf の法則が現れています．

という関係が成り立っています．式(3.8)は，変形すると $\log fr = b$ となりますから，$e^b = c$ とおけば

$$fr = c \tag{3.9}$$

あるいは，

$$f = c \cdot \frac{1}{r} \qquad (\textbf{Zipf の法則}) \tag{3.10}$$

という関係が成り立つことになります．すなわち，単語は頻度順に順位が $r = 1, 2, 3, 4 \cdots$ になるほど，その相対的な頻度は $1, 1/2, 1/3, 1/4, \cdots$ と減っていく，ということです．この関係は，他のどんなテキストに対してもほぼ成り立つことが知られています．図 3.4(b) に，Brown コーパス `brown.txt` に対して同様に計算したプロットを示しました．ここでも，同じ法則が成り立っていることがわかります．この法則は，1930 年代にこの法則を体系化したハーバード大学の言語学者Zipf（ジップ）*19 の名前をとって，**Zipf の法則**とよばれています．Zipf の法則は，言語に限らず社会全般で広く成り立つ**巾乗則**（べき）として認知されており，たとえば

*19　George Kingsley Zipf (1902–1950) は，1935 年に[53]で Zipf の法則を示しました．ただし，この法則は Zipf が最初に発見したわけではなく，1916 年にはフランスの速記者 Jean-Baptiste Estoup によって発見されていました[54, 55]．最近になり，Estoup の孫にあたる方によって，この事情についての論文が出版されています[56]．

都市の大きさ, 収入の分布, 地震の規模と頻度など多くの実際の現象が巾乗則に従うことが知られています[57]. 言語を中心としたこうした巾乗則については, [58]で詳しく論じられていますので参照してください.

この Zipf の法則によれば, 語彙が 10,000 個であれば, 最もよく現れる単語の確率と最も現れにくい単語の確率の比は, 1:1/10000 で 10,000 倍にもなる, ということになります. 文字の場合は, 2.1 節で見たように英語では最も頻度の高い文字 'e' と最も低い文字 'z' の確率の比は 150 倍くらいでしたから *20, 語彙の多い単語の場合は, その差は圧倒的に大きくなっていることがわかります.

なお, 単語の確率を式(2.1)のように相対頻度で計算する場合, 確率は頻度に比例しますから, 式(3.10)から最も頻度の高い単語の確率を $p_1=q$ とおくと, 2 番目に高い確率 p_2, 3 番目に高い確率 p_3, $\cdots$ はほぼ

$$p_1 = q,\ p_2 = \frac{q}{2},\ p_3 = \frac{q}{3},\ p_4 = \frac{q}{4},\ \cdots$$

です. よって, その総和は

$$\begin{aligned} p_1 + p_2 + p_3 + p_4 + \cdots &= q\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots\right) \\ &= q\sum_{r=1}^{\infty}\frac{1}{r} \end{aligned} \tag{3.11}$$

となります. しかし, 式(3.4)でみた調和級数 $\sum_{r=1}^{\infty} 1/r^{\alpha}$ は $\alpha=1$ のとき無限大に発散しますから, 式(3.11)の確率の和は 1 にならなくなってしまいます. よって級数が収束するためには, 少なくともある順位からは $\alpha>1$ になっているはずです. 実際に図 3.4(a) でもみられるように, 一般に $r\to 0$ と $r\to\infty$ では巾乗則の傾き α が異なっており, この現象は double power-law [59]とよばれています. この説明として, 語彙を核となる語彙と周辺の語彙に分ける生成モデル[60]などが研究されています *21.

---

*20 英語の ASCII 文字は 7 ビットで表され, 記号を含めて $2^7=128$ 文字ありますから, Zipf の法則からもこの差は妥当なオーダーだということがわかります.

*21 数学的には, 最近オックスフォード大学の Caron らによって, Lévy 測度 $\rho(w) = 1/\Gamma(1-\sigma)w^{-1-\tau}\gamma(\tau-\sigma, cw)$ ($\gamma()$ は第一種不完全ガンマ関数) をもつ一般化 BFRY (Bertoin-Fujita-Roynette and Yor) 過程を考えると, これを和が 1 になるように正規化した正規化 GBFRY 過程がこの double power-law をもち, 言語に非常によく当てはまることが示されています[61].

コラム：単語の前処理について

ここまでの実行例では, 空白で区切られたすべての文字列を単語として扱いましたが, 実際にはテキストには数字や特殊文字 (記号や罫線など) が入っており, これらをそれぞれ別の単語として扱うと, 特に数字によって語彙が爆発的に増えてしまいます. また, 英語では `I'm` や `Alice's` はそれぞれ, `I 'm`, `Alice 's` と分割して, `I` や `Alice` を単語とみなすのが望ましいでしょう.

そこで, 一般にたとえば数字を`#`で置き換えたり, 特殊文字を削除したり, `aaaaaaa` のような同じ文字の連続を 3 文字に縮約して `aaa` とするなどの前処理を行います. こうすると, たとえば `No.123` は `no.###`に, `2021.3.28` は `####.#.##` に, `Gooooooooal` は `goooal` になります. また, 大文字と小文字の違いで別の単語と認識されないよう, ここに示したようにすべて小文字に直すこともよく行われる処理の一つです.*22

前処理には決まった手続きがあるわけではありませんが, 英語の場合は教科書 FSNLP [38]のサイトにある `strip.sed`*23 などを使うのが簡単です. 日本語の場合は, 後で紹介する mecab-NEologd のサイト *24 に正規化処理についてまとめられているほか, この処理を pip でインストールできるパッケージにした `neologdn`*25 が公開されています. 表 3.2 は, こうした前処理を行った上で単語を数えたものです.

また, Python の spaCy や R の quanteda といったパッケージに前処理を任せることも可能です.

3.3▽ 単語の統計的フレーズ化

*22　多くの言語で何を大文字にするかには規則性があるため, 文脈を利用して単語の各文字を小文字から適切に大文字にする分類器を教師あり学習することは容易です. よって, 小文字にしても本質的な情報量が落ちることはあまりありませんが, たとえば `US` を `us` として処理すると誤ることもありますので, すべて小文字化することにはリスクもあることに注意してください.

*23　`https://nlp.stanford.edu/fsnlp/statest/sed.strip`

*24　`https://github.com/neologd/mecab-ipadic-neologd/wiki/Regexp.ja`

*25　`https://github.com/ikegami-yukino/neologdn`
`https://yukinoi.hatenablog.com/entry/2015/10/11/205006`

3.1 節で述べたように, 英語では “New York”, “with respect to”, 日本語では「基本 的」,「富嶽 三 十 六 景」のように, 空白で区切られていても, 実際は一つの単語として働く表現は多く存在します. また,「御 樋 代 木 奉 曳式」*26 のように特殊な言葉のために形態素解析が失敗して多くの単語に分割されてしまっている例もありますし,「高 エネルギー リン 酸 化合 物」「お ジャ 魔女 どれ み #」*27 のような長い単語列を固有名詞として認識したい場合もあるでしょう. このような場合には, どうすればいいのでしょうか.

日本語の場合, 最も簡単な方法は, 多くの新語や固有名詞に対応し, 人手で更新されている MeCab 用の辞書 mecab-ipadic-NEologd [62] *28 を用いることです. これには多くの固有表現が含まれており, 一般的な語については一語として認識して形態素解析されます. より一般には, 認識したい上記のような固有表現の正解データを多数, 人手で用意しておけば, そこから教師あり学習によって, 新しいテキストに対して該当個所を固有表現として認識してくれる識別器を学習する**固有表現認識** (Named Entity Recognition, NER) とよばれる方法でフレーズを認識することができます (本書では正解データが必要な教師あり学習は基本的に扱いませんので, 興味のある方は[10, Sec.5.5]などを参照してください).

ただし, 前者の辞書は人手で更新されているものですから, 有名な固有名詞はカバーされているものの,「t 細胞受容体」「ディビダーク式カンチレバー工法」のような専門的な名前にすべて対応することは, 原理的にできません. また, 後者の NER はあくまで正解データとして与えた表現およびそれに近い表現を認識するものですから, あらゆるフレーズに対応するには, 厖大な量の「正解データ」を逐次作る必要があり, その際の基準も一意とは限らないために現実的ではありません.

しかし, よく考えると, “San” の後には必ず “Francisco” が続くのであれば, “San Francisco” を一つの単語として認識してもよいはずです. 同様に,「宮内」

*26 御樋代木奉曳式 (みひしろぎほうえいしき) は, 木曽の山中から切り出された御料木が用材として伊勢神宮に運ばれる儀式とのことです.

*27 この「どれ み」の例のように, 未知の固有名詞に対しては形態素解析自体が失敗することがよく起こります.

*28 `https://github.com/neologd/mecab-ipadic-neologd`

の後には「庁」が続く確率が非常に高く, また「楽部」の前は「宮内 庁」であることが非常に多いのであれば, 「宮内庁」「宮内庁楽部」をそれぞれ単語とみなしてもよいでしょう.

このような考え方に基づき, 3.5.2 節で説明する単語のベクトル化の方法である有名な Word2Vec を発明した Mikolov ら [63]は, 空白で区切られた単語をそのままベクトル化するのではなく, 文脈に応じて "and new_york city council .." のように '_' によって単語を繋げて自動的にフレーズ化してから Word2Vec を計算する方法を示しました. *29

この場合, 単語 v と単語 w を繋げてフレーズにするかどうかは, 次のスコアによって統計的に決定します.

$$\mathrm{score}\,(v,w) = \frac{n(v,w)-\delta}{n(v)\times n(w)} \tag{3.12}$$

ここで, $n(v,w)$ および $n(v), n(w)$ はそれぞれ単語バイグラムおよびユニグラムの頻度で, δ は頻度 $n(v,w)$ の小さいバイグラムのスコアを下げるための割り引き係数です. たとえば $v=\mathtt{san}$, $w=\mathtt{francisco}$ で, $n(v,w)=100$, $n(v)=100$, $n(w)=120$ だったとき *30, $\delta=10$ であれば

$$\mathrm{score}\,(\mathtt{san},\mathtt{francisco}) = \frac{100-10}{100\times 120} = 0.0075$$

になります. 一方 $v=\mathtt{read}$, $w=\mathtt{books}$ で, $n(v,w)=100$, $n(v)=1000$, $n(w)=500$ であれば,

$$\mathrm{score}\,(\mathtt{read},\mathtt{books}) = \frac{100-10}{1000\times 500} = 0.00018$$

となり, "read_books" より "san_francisco" のスコアの方が約 40 倍も大きいことになります. よって, たとえば閾値を 0.001 として, 隣りあった単語のスコアが閾値より大きい場合に単語を連結すれば, "san_francisco" のようなフレーズを自動的に認識することができます.

---

*29　この方法は, すでに 1992 年には IBM の Brown ら[64]によって提案されています.

*30　「フランシスコ修道会」のような表現もありますから, 少数ですが, francisco は他で用いられる場合もあります.

フランクリン・ルーズベルト の とっ た [ニューディール_政策] の [一環_として] 、 1935 年 に 連邦 [社会_保障] 法 が 制定 [さ_れ] 、 失業 保険 と 老齢 年金 が 整備 [さ_れ] た
一方 の 越後 で は 、 [守護_代] ・ 長尾 氏 により 国内 が 統一 [さ_れ] 、 氏康 により 北関東 から 駆逐 [さ_れ] た [関東_管領] の [上杉_憲政] から [山内_上杉] 家 の 家督 と 管領 職 を 継承 [し_た] [上杉_謙信] は 北関東 で 氏康 と 対決 し 、 信濃 で は 北信 勢力 を 後援 [し_て] 信玄 と 対決 する 二 正面 作戦 を 展開 し [て_い] た

図 3.5: 正規化自己相互情報量 (NPMI) に基づいて 2 単語を統計的にフレーズ化した例. 認識されたフレーズを [] で示しました. 学習には日本語版 text8 コーパスを使用しています.

ただし, Mikolov らの論文 [63]で提案されている式(3.12)のスコアには注意が必要です [*31]. テキスト全体の長さを N とすると, バイグラム (v, w) およびユニグラム v, w の確率はそれぞれ

$$p(v,w) = \frac{n(v,w)}{N}, \quad p(v) = \frac{n(v)}{N}, \quad p(w) = \frac{n(w)}{N} \tag{3.13}$$

ですから, $n(v,w) = N \cdot p(v,w)$, $n(v) = N \cdot p(v)$, $n(w) = N \cdot p(w)$ を式(3.12)に代入すると,

$$\mathrm{score}\,(v,w) = \frac{n(v,w) - \delta}{n(v) \times n(w)} = \frac{N \cdot p(v,w) - \delta}{N \cdot p(v) \times N \cdot p(w)} = \frac{p(v,w) - \delta/N}{p(v) \times p(w)} \cdot \frac{1}{N} \tag{3.14}$$

となり, このスコアは学習テキストの長さNに依存します. さらに, 割り引き係数δの大きさもNによって変えなければならない, ということがわかります. テキストを固定して閾値を探索するのであれば問題ありませんが [*32], スコアに絶対的な意味がないために探索が難しく, 違うテキストを同じ基準で前処理することもできなくなってしまいます. この欠点はスコアを N 倍すれば解消しますが, その場合も δ/N の部分は残るため, 適切な δ の設定は困難になります.

ここで $\delta = 0$ としてみると, 式(3.12)は式(3.14)より

*31 以下の内容は, 本書のオリジナルです.
*32 実際には, フレーズ化を数パス繰り返す中で単語が結合され, N の値も変わってきてしまいますので, もとの方法では適切な閾値の設定はさらに困難になります.

$$\text{score}\,(v,w) = \frac{p(v,w)}{p(v)\times p(w)}\cdot\frac{1}{N} \tag{3.15}$$

となり, これは5章でも説明する**自己相互情報量** (Pointwise Mutual Information, PMI)

$$\text{PMI}\,(v,w) = \log\frac{p(v,w)}{p(v)p(w)} \tag{3.16}$$

と, 本質的に同じ意味を持っていることがわかります. 式(3.16)の PMI は, v と w が共起する確率 $p(v,w)$ が, v と w が独立だった場合の確率 $p(v)p(w)$ と比べて何倍なのか (の対数) を表しています. これは統計的に v と w の相関を見るためには理想的な量で, 情報理論に基づいて 1989 年にベル研究所の Church ら[65]によって提案されました.

ただし, PMI には

- v や w が非常に低頻度で, $p(v)$ や $p(w)$ がきわめて小さい場合に式(3.16)が非常に大きくなってしまう
- 最大値・最小値が v,w によって異なり, 一定ではない

という欠点があります. そこで, 5 章でも説明するように, PMI をその最大値である, $-\log p(v,w)$ で割って正規化した**正規化自己相互情報量** (Normalized PMI, **NPMI**) [66]

$$\text{NPMI}\,(v,w) = \log\frac{p(v,w)}{p(v)p(w)}\Big/\Big(-\log p(v,w)\Big) \tag{3.17}$$

を用いることを考えてみましょう. この NPMI は $p(v)$ や $p(w)$ が小さくても値がインフレせず,

$$-1 \le \text{NPMI}\,(v,w) \le 1 \tag{3.18}$$

の値をとり, v と w が完全に相関しているとき 1, 完全に逆相関しているとき -1 の値をとるという, 大変よい性質を持っています. 式(3.17)は, 頻度 $n()$ を使えば

$$\text{(3.19)}\quad \mathrm{NPMI}\,(v,w) = \log\frac{p(v,w)}{p(v)p(w)}\Big/\Big(-\log p(v,w)\Big)$$

$$= \log\left(\frac{n(v,w)}{\cancel{N}}\cdot\frac{\cancel{N}}{n(v)}\cdot\frac{N}{n(w)}\right)\Big/\left(-\log\frac{n(v,w)}{N}\right)$$

$$\text{(3.20)}\quad = \frac{\log N + \log n(v,w) - \log n(v) - \log n(w)}{\log N - \log n(v,w)}$$

と表すことができます．このとき，式(3.12)の**ヒューリスティックな割り引き係数 δ は不要になっている**ことに注意してください．この計算は，サポートサイトにある `phraser.py` を使って，

```
% phraser.py ja.text8.txt output 4
```

のように実行すると簡単に行うことができます．

この NPMI が閾値，たとえば 0.5 以上になった 2 単語をフレーズとみなして日本語 text8 に適用した例を図 3.5 に示しました．ほとんどの語はそのままですが，NER を使わなくても教師なしで，「ニューディール_政策」「上杉_謙信」「さ_れ」などがフレーズとして自動的に認識されていることがわかります．

この方法は隣り合う 2 単語を結合するものですが，この出力を入力として再度フレーズ化すれば，最大で 4 単語までのフレーズが得られます．同様にしてこのフレーズ認識を n パス動かせば，2^n 語までのフレーズを計算することができます．表 3.3 に，こうして 4 パス (=最大 16 単語まで) を日本語 text8 コーパスに適用して得られたフレーズの例を示しました．

表からわかるように，「である」「となっていた」「キリスト教徒」といった，日本語によくあるフレーズや固有名詞が教師なしで正しく認識されており，時には「福島第一原子力発電所」といった長い固有名詞も自動的にフレーズ化されていることがわかります．テキストに対してこうした前処理を行うことで，意味的内容をより正しく反映させることができます．

しかし一方で，たとえば「逆転写酵素」が「酵素」の一種だという情報は失われてしまうため，こうしたフレーズ化は，フレーズの頻度が充分高い場合にのみ行った方がよいでしょう．上記の `phraser.py` では，デフォルトではバイグラム頻度が 10 以上の場合にのみフレーズ化を行う設定になっており，閾値も含めてオプションで変えることができます．使い方は，

表 3.3: 日本語版 text8 から NPMI に基づいて統計的に認識されたフレーズとその頻度. _は, MeCab によるもとの単語区切りを表しています. 4 パスで計算したため, 最大で 16 単語までがフレーズの候補になっています. NPMI の閾値は 0.5 としました.

頻度	フレーズ
107101	し_た
70074	で_ある
48360	さ_れ_た
31219	て_いる
28473	し_て_いる
23371	さ_れ
:	
1122	第_二_次_世界_大戦
1094	と_なっ_て_い_た
1092	最終_的
1088	基本_的
:	
100	について_述べる
100	キリスト_教徒
100	陸上_競技_選手
100	ゆう_ちょ_銀行
:	
45	全国_高等_学校_野球_選手権_大会
37	お_ジャ_魔女_どれ_み
32	福島_第_一_原子力_発電_所_事故
27	ほっ_か_ほっ_か_亭
26	福島_第_一_原子力_発電_所
23	連合_国軍_最高_司令_官_総_司令_部
19	学研_奈良_登美_ヶ_丘_駅
16	国際_連合_安全_保障_理事_会
10	ソビエト_社会_主義_共和_国_連邦
10	自転車_競技_選手権_大会
10	工学部_機械_工学科
10	合衆国_最高_裁判所
10	衆議院_予算_委員_会
10	大学院_工学_研究_科

```
% phraser.py
```

をそのまま実行するか, スクリプトの中身を読んでみてください.

3.4 単語 n グラム言語モデル

単語についても, 2.5 節の文字の場合と同じように単語 n グラム言語モデルを考えることができます. 現在は文の確率を計算するには, LSTM や Transformer などの深層学習モデルを利用するのが一般的ですが [*33], 本書で扱うさまざまな統計モデルの基礎になりますので, 単純で理由のわかっている, この最も基本的なモデルについて考えていくことにしましょう.

単語が直前の単語だけに依存するバイグラム言語モデルを考えるとすると, 文 $s =$「銀河 鉄道 の 夜」の確率は式(2.45)と同様に,

$$p(s) = p(\text{銀河} \mid \hat{}\,)\, p(\text{鉄道} \mid \text{銀河})\, p(\text{の} \mid \text{鉄道})\, p(\text{夜} \mid \text{の})\, p(\$ \mid \text{夜}) \tag{3.21}$$

と表すことができます. ここで ^, $ はそれぞれ, 2.5 節で導入した文頭および文末を表す特別な単語です.

このバイグラム頻度を数えるには, Python ではたとえば, 3.1 節で単語に分割したテキストを使って

```
from collections import defaultdict
def parse (file):
    EOS = "_EOS_"
    freq = {}
    with open (file, 'r') as fh:
        for line in fh:
            words = line.rstrip('\n').split() # 空白で分割
            words.insert (0, EOS); words.append (EOS)
            T = len(words)          # ↑文頭/文末文字を追加
            for t in range(T-1):
                w = words[t]
                v = words[t+1]
                if not (w in freq): # freq[w] がなければ作成
                    freq[w] = defaultdict(int)
                freq[w][v] += 1     # 頻度に 1 を加える
    return freq
```

*33 ただし, こうした深層学習モデルがどのように単語を予測しているのかは, まだ充分わかっていないのが現状です.

のように関数を定義してから,

```
    freq = parse ("ginga.split.txt")
    freq
 ⇒ {'_EOS_': defaultdict(int,
                {'銀河': 1,
                 '一': 1, ...
    {'銀河': defaultdict(int,
                {'鉄道': 2,
                 '帯': 1,
                 'を': 2,
                 'は': 1,
                 'の': 11,
                 'が': 2,
                 'ステーション': 5,
                 'だ': 1}), ...
```

のように実行すれば, `freq[w][v]` にバイグラムの頻度 $n(w,v)$ を格納することができます. たとえば上の場合は,

```
    freq["銀河"]["ステーション"]
 ⇒ 5
```

のようになります.

式(3.21)のそれぞれのバイグラムの条件つき確率は, 頻度を表す関数 $n()$ を使って, 最も簡単には式(2.4)のような最尤推定値

$$\widehat{p}(v|w) = \frac{n(w,v)}{\sum_v n(w,v)} = \frac{n(w,v)}{n(w)} \tag{3.22}$$

で求めることができるのでした. ここで $n(w,v)$ は単語バイグラム wv の出現した頻度, $n(w)=\sum_v n(w,v)$ は単語 w の頻度です.

ただし, 式(3.22)を使った単語バイグラム確率の計算には, すぐに問題があることがわかります. ここまでにみたように, Zipf の法則から頻度のほとんどは一部の単語に集中しており, 大部分の単語は低頻度なのでした. すると, たとえば「銀河」の次に「旅行」が現れる確率は, 式(3.22)に従えば

$$p(\text{旅行} \mid \text{銀河}) = \frac{n(\text{銀河}, \text{旅行})}{n(\text{銀河})} \tag{3.23}$$

となりますが,『銀河鉄道の夜』の中では $n(\text{銀河}, \text{旅行}) = 0$ なので,

$$p(\text{旅行} \mid \text{銀河}) = \frac{n(\text{銀河}, \text{旅行})}{n(\text{銀河})} = \frac{0}{n(\text{銀河})} = 0 \tag{3.24}$$

となり,「銀河旅行」の確率は式(2.20)の確率の連鎖則から,

$$p(\text{銀河 旅行}) = \underbrace{p(\text{旅行} \mid \text{銀河})}_{=0} \cdot p(\text{銀河}) = 0 \tag{3.25}$$

となり, 0 になってしまいます. こうした問題を, 2.5.2 節で述べたように**ゼロ頻度問題**と呼びます.

文字の場合は種類が少ないため, 文字バイグラムではゼロ頻度問題はあまり深刻になりませんでしたが [*34], 単語は種類が多いため, バイグラムでも, ほとんどの確率が 0 になってしまうという問題が生じます.

ゼロ頻度問題を避けるために, 2 章の文字 n グラムモデルでは式(2.51) の**加算平滑化**という手法を用いました. 語彙の総数を V とおくと, これは式(3.22) の代わりに, すべての頻度に小さな値 α を足して

$$p(v|w) = \frac{n(w,v)+\alpha}{\sum_{v=1}^{V}\big(n(w,v)+\alpha\big)} = \frac{n(w,v)+\alpha}{n(w)+V\alpha} \tag{3.26}$$

とするものです. こうすると, $n(w,v)=0$ の場合でも, 確率は

$$p(v|w) = \frac{\alpha}{n(w)+V\alpha}$$

となり, 出現しない語にも, α に比例する一定の確率を割り当てることができます.

ところが, 単語の場合は式(3.26)を使ってもまだ問題があることがわかります. たとえば, 単語 w と v は「銀河 鉄道」のように常に wv の形で出現しており, $n(w,v)=10$, $n(v)=n(w)=10$ だったとしましょう. $\alpha=0.01$, $V=10000$ とすると, 式(3.26)は

*34 日本語や中国語などの漢字圏では, 文字の種類も非常に多いため, 文字バイグラムの場合でもゼロ頻度問題は深刻になります.

$$p(v|w) = \frac{n(w,v)+\alpha}{n(w)+V\alpha} = \frac{10+0.01}{10+10000\cdot 0.01} = \frac{10.01}{110} = 0.091 \tag{3.27}$$

となります．つまり，w の後にはつねに v が続くにもかかわらず，$p(v|w)$ はたった 0.09 にしかならず，それ以外の確率が $1-0.091=0.909$ で 91%もある，という結果になってしまうのです．

これはもちろん，「現れなかった単語も含め，すべての単語の頻度に同じαを足す」ということが原因です．「群」のような頻度の高い語も，「パセリ」のような頻度の低い語も同じ α が足されるため，たまたま「銀河」の後に続いたことがなければ「銀河群」と「銀河パセリ」が同じ確率を持つことになり，それらの確率の総和である式(3.26)の $V\alpha$ の部分が非常に大きくなってしまうからです．実際に，式(3.26)を変形すると，

$$\begin{aligned} p(v|w) &= \frac{n(v,w)+\alpha}{n(w)+V\alpha} = \frac{n(v,w)}{n(w)+V\alpha} + \frac{\alpha}{n(w)+V\alpha} \\ &= \frac{n(w)}{n(w)+V\alpha}\cdot\frac{n(v,w)}{n(w)} + \frac{V\alpha}{n(w)+V\alpha}\cdot\frac{1}{V} \\ &= \lambda\cdot\widehat{p}(v|w) + (1-\lambda)\cdot\frac{1}{V} \qquad \left(\lambda = \frac{n(w)}{n(w)+V\alpha}\right) \end{aligned} \tag{3.28}$$

となります．これから，式(3.26)の確率は式(3.22)の最尤推定値 $\widehat{p}(v|w)$ と，**すべての単語に一様な確率を与える** $p_0(v)=\dfrac{1}{V}$ を比率 $\lambda:(1-\lambda)$ で補間した確率になっていることがわかります．*35 これは，$\lambda=\dfrac{n(w)}{n(w)+V\alpha}$ が小さい，すなわち $n(w)$ が小さいほど，確率の推定値が $\dfrac{1}{V}$ に近づくことを意味しています．つまり，「銀河」自体の出現回数が少なければ，「銀河 の」も「銀河 パセリ」も同じくらい出現しやすいと仮定していることになるわけです．

言うまでもなく，これは誤りです．文脈となる語 w と予測したい語 v を分けて考えると，v に対応する α の値は，大まかには v の確率 $p(v)$ に比例する値 α_v とするのがよいでしょう．ただしよく考えると，単に $p(v)$ に比例するのではな

*35　このように，複数の確率分布 (この場合は $\widehat{p}(v|w)$ と $p(v)=1/V$) を重み (λ と $1-\lambda$) つきで混ぜ合わせたモデルを，**混合モデル**といいます．

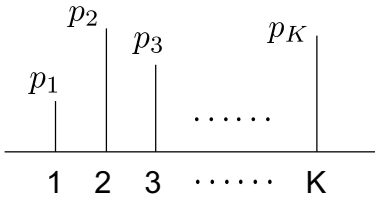


図 3.6: K 次元の多項分布 $\mathbf{p}$. 確率の和は $\sum_{k=1}^{K} p_k = 1$ になっています.

く, たとえ同じ頻度でも, 多くの語の後に続きやすい「前」のような語の α_v は大きく, 他の語の後に続きにくい「あの」のような語の α_v は小さくするべきだと考えられます.[*36] そして, 決めるべきパラメータは $\alpha_1, \alpha_2, \cdots, \alpha_V$ で, この場合 10,000 個もあります. この $\alpha_1, \alpha_2, \cdots, \alpha_V$ はどうやって決めればよいのでしょうか.

こういった複雑な問題を解くには, 今までのような発見的な方法では限界があります. そこで, より原理的に考えてみることにしましょう.

3.4.1 ディリクレ分布

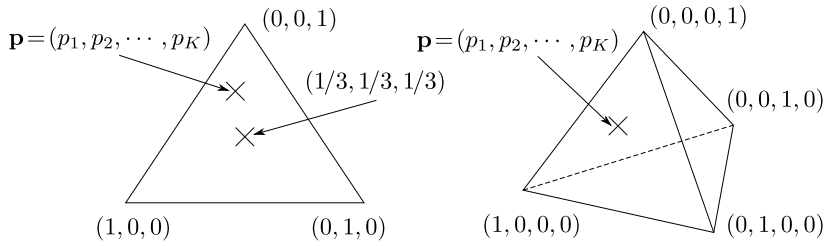
これまで, 式(3.26)のような確率の由来については考えず, 単に「頻度やそれに値を足したものを和が 1 になるように正規化する」ことしか行ってきませんでした. 式(3.26)のような確率は, すべての単語 $v = 1, 2, \cdots, V$ について考えると総和が 1 になる**確率分布**ですから, そもそも確率分布を生み出すことのできる**確率モデル**について考えてみましょう.

いま, 図 3.6 のような K 次元の確率分布を

$$\mathbf{p} = (p_1, p_2, \cdots, p_K) \qquad \left(p_1, p_2, \cdots, p_K \geq 0, \sum_{k=1}^{K} p_k = 1 \right) \tag{3.29}$$

とします. K は次元の数で, 言語の場合には K は実際には 10,000 や 100,000 といった大きな値になります. 確率 $p_1, p_2, \cdots, p_K$ を直接指定するこの確率分布は,

*36 『銀河鉄道の夜』では, 「前」と「あの」の出現頻度はどちらも 38 回で同一です.



(a) $K=3$ の場合　　(b) $K=4$ の場合

図 3.7: 単体とその上の多項分布 $\mathbf{p}$. 単体の各端点は, そのカテゴリだけが確率 1 で出る確率分布になっています.

もっとも基本的な分布で, 正確には**多項分布**または離散分布とよびます.[*37]

たとえば $K=3$ のとき,

$$\mathbf{p} = (0.7, 0.1, 0.2)$$

は $\mathbf{p}$ の 1 つの例です. より一般に $\mathbf{p}=(p_1, p_2, p_3)$ は, これをベクトルとみなせば, 図 3.7(a) のように正三角形の内部にあると考えることができます. こうした図形を, **単体** (simplex) といいます. $\mathbf{p}=(1,0,0)$, $(0,1,0)$, $(0,0,1)$ の各分布はそれぞれ, この単体の 3 つの角に対応し, $\mathbf{p}=\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ は単体の中心に対応しています. なお, $K=4$ の場合は, 単体は図 3.7(b) のような正四面体になります.

$\mathbf{p}=(p_1, p_2, \cdots, p_K)$ が与えられたとき, 観測値の確率は p_k の積で簡単に求めることができます. たとえば,

$$\mathbf{p}=(p_1, p_2, p_3, p_4)$$

のとき, $\mathbf{p}$ に従ってランダムに選んだ結果が

$$Y = (4, 2, 1, 1, 2)$$

だったとすれば [*38], Y の確率は

---

*37　多項分布は, 本来は二項分布の拡張で $\mathbf{p}$ から N 回サンプルした結果の確率を与えるものです. ただし, 離散的な分布にはポアソン分布など他の分布も含まれるため, それと区別して $N=1$ の場合も多項分布ということがあり, 本書でもこの表記を用います.

*38　ここでは, Y にテキストのように順番を考えています. 順番を考えない場合は, 可能な組み

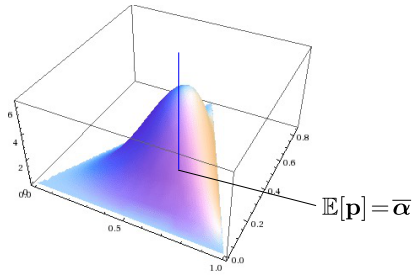


図 3.8: 単体上のディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ とその期待値 $\mathbb{E}[\mathbf{p}]=\overline{\boldsymbol{\alpha}}$.

$$
\begin{aligned}
p(Y|\mathbf{p}) &= p_4 \cdot p_2 \cdot p_1 \cdot p_1 \cdot p_2 = p_1^2 \cdot p_2^2 \cdot p_3^0 \cdot p_4^1 \\
&= \prod_{k=1}^{4} p_k^{n_k}
\end{aligned}
\tag{3.30}
$$

となるわけです. 当たり前ですね. ここで n_k は Y の中で k が出た回数で, この例では $(n_1, n_2, n_3, n_4) = (2, 2, 0, 1)$ となります.

こうした $\mathbf{p}$ を生成する単体上の最も簡単な分布として, 次の式で表される**ディリクレ分布** (Dirichlet distribution) があります. [*39]

$$
\mathrm{Dir}(\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} p_k^{\alpha_k - 1} \qquad (\textbf{ディリクレ分布 (簡易版)})
\tag{3.31}
$$

図 3.8 に, この分布の形を示しました. $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_K)$ はこの分布のパラメータで, $\alpha_k \geq 0$ $(k = 1, 2, \cdots, K)$ は非負の実数です [*40]. $\mathbf{p}$ はそれ自身確率分布ですから, ディリクレ分布は「確率分布を生成する確率分布」ということになります. ディリクレ分布から生成される $\mathbf{p}$ の期待値 $\mathbb{E}[\mathbf{p}]$ は, $\boldsymbol{\alpha}$ を和が 1 になるように正規化した

合わせの総数である多項係数 $\binom{5}{2\ 2\ 0\ 1} = \frac{5!}{2!\,2!\,0!\,1!}$ がかかることになりますが, この係数はパラメータ $\mathbf{p}$ を含んでいませんので, $\mathbf{p}$ に関する推定値は同じになります.

*39 この名前は, 正規化定数となる式(3.40)の積分を示した, 旧フランス領で生まれたドイツの数学者 Lejeune Dirichlet (1805–1859) にちなむものです [67].

*40 $\alpha_k = 0$ のときは, 対応する p_k はつねに 0 であると約束します.

(3.32)

$$\mathbb{E}[\mathbf{p}] = \overline{\boldsymbol{\alpha}} = \left(\frac{\alpha_1}{\sum_k \alpha_k}, \frac{\alpha_2}{\sum_k \alpha_k}, \cdots, \frac{\alpha_K}{\sum_k \alpha_k} \right) \quad (\textbf{ディリクレ分布の期待値})$$

となります.

ディリクレ分布からのサンプルは, Python では `numpy.random.dirichlet()` で作ることができます. または, ガンマ分布 $\mathrm{Ga}(\alpha_k, 1)$ からのサンプル *41

(3.33)
$$\gamma_k \sim \mathrm{Ga}(\alpha_k, 1) \quad (k = 1, 2, \cdots, K)$$

が得られれば, それを和が 1 になるように正規化することで, ディリクレ分布からのサンプル

(3.34)
$$\mathbf{p} = \frac{1}{\sum_k \gamma_k} (\gamma_1, \gamma_2, \cdots, \gamma_K) \sim \mathrm{Dir}(\alpha_1, \alpha_2, \cdots, \alpha_K)$$

を計算することができます. サポートページにある `dirichlet.py` を, たとえば

```
% dirichlet.py 0.5 5
```

のように実行すれば, 5 次元でパラメータがすべて 0.5 のディリクレ分布

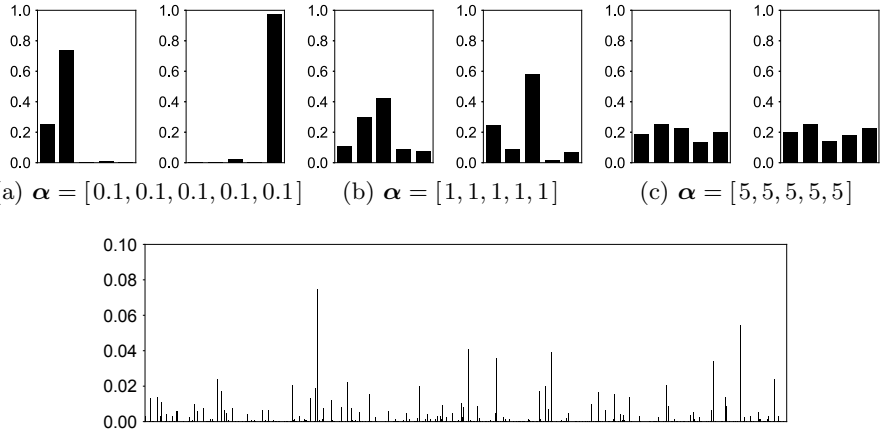
$$\mathrm{Dir}(0.5, 0.5, 0.5, 0.5, 0.5)$$

から生成されたランダムな多項分布 $\mathbf{p}$ をプロットすることができますので, 試してみましょう. 図 3.9 に, いくつかの $\boldsymbol{\alpha}$ の値について, こうして $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成した多項分布 $\mathbf{p}$ の例を示しました. $\alpha_k = 1$ の場合は $\mathbf{p}$ は自由な形になりますが, $\alpha_k > 1$ の場合は $\mathbf{p}$ は期待値である $\overline{\boldsymbol{\alpha}}$ に近く, $\alpha_k < 1$ の場合は $\mathbf{p}$ は少数の p_k の値だけが大きく, 他がほとんど 0 に近い疎な分布となることがわかります. 言語の場合は, ほとんどはこの場合に対応しています.

ガンマ関数とディリクレ分布

---

*41　この 1 はスケールを表しているだけですので, 同一であれば別の値でも問題ありません. ガンマ分布からのサンプルは, Python では `numpy.random.gamma()` で作ることができます. 自分で $[0, 1)$ の乱数からガンマ関数に従う乱数を計算する方法は複雑ですので, 必要な方は乱数生成の専門書[68]を参照してください. サポートページに, 筆者が使っている C 言語による実装 `gamma.{c,h}` が置いてあります.



(a) $\boldsymbol{\alpha} = [0.1, 0.1, 0.1, 0.1, 0.1]$ (b) $\boldsymbol{\alpha} = [1, 1, 1, 1, 1]$ (c) $\boldsymbol{\alpha} = [5, 5, 5, 5, 5]$

(d) $\boldsymbol{\alpha} = [0.05, 0.05, \cdots, 0.05]$ (1000 次元)

図 3.9: ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ からランダムにサンプルした多項分布 $\mathbf{p}$ の例.

式(3.31)では比例 $\propto$ を使ってディリクレ分布を示しましたが, 正確に書くと, ディリクレ分布の定義は

$$\mathrm{Dir}(\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} \qquad (\textbf{ディリクレ分布 (正式版)}) \tag{3.35}$$

となります. この定義式(3.35)は一見いかつい形をしていますが, ガンマ関数 $\Gamma(x)$ を含む前半部分は, 可能な $\mathbf{p}$ 全体についての積分を 1 にするための正規化定数で, 本質的には, ディリクレ分布は式(3.31)で表される簡単な分布であることに注意してください. $\Gamma(x)$ $(x \in \mathbb{R})$ は階乗関数 $x! = x(x-1)(x-2)\cdots 1$ の連続値への一般化とみることができる関数で,

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \qquad (\textbf{ガンマ関数}) \tag{3.36}$$

で定義されます. 式(3.36)を部分積分することで,

$$\Gamma(x+1) = x\Gamma(x) \tag{3.37}$$

を示すことができますので, 確かめてみましょう (→演習問題 1). 式(3.37)から,

x が整数であれば

$$\begin{aligned}\Gamma(x) &= (x-1)\Gamma(x-1) = (x-1)(x-2)\Gamma(x-2) \\ &= (x-1)(x-2)\cdots\cdot 2\cdot 1 \\ &= (x-1)!\end{aligned} \tag{3.38}$$

となります. x が整数でない場合は, x を超えない最大の整数を $n=\lfloor x\rfloor$ とおき, $x=n+\alpha$ と書けば

$$\begin{aligned}\Gamma(x) &= (n+\alpha-1)\Gamma(n+\alpha-1) \\ &= (n+\alpha-1)(n+\alpha-2)\cdots\cdot(\alpha+1)\cdot\alpha \\ &= (x-1)(x-2)\cdots\cdot(\alpha+1)\cdot\alpha\end{aligned} \tag{3.39}$$

となり, 確かにいずれの場合も, $\Gamma(x)$ は階乗 $(x-1)!$ の一般化になっていることがわかります.

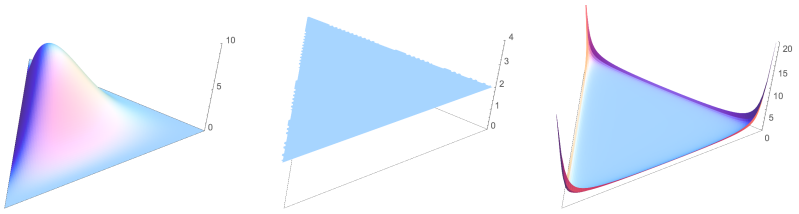
式(3.35)の正規化定数は, この $\Gamma(x)$ を使って

$$\begin{aligned}\int\prod_{k=1}^{K} p_k^{\alpha_k-1}d\mathbf{p} &= \int_0^1\int_0^1\cdots\int_0^1 p_1^{\alpha_1-1}p_2^{\alpha_2-1}\cdots p_K^{\alpha_K-1}dp_1dp_2\cdots dp_K \\ &= \frac{\prod_k\Gamma(\alpha_k)}{\Gamma(\sum_k\alpha_k)}\end{aligned} \tag{3.40}$$

の積分から得られるものです. この積分は, 直感的には, 図 3.8 のような単体上の曲面の下の体積を求めることに相当しています. 式(3.40)の積分は大学教養範囲の数学 (解析) を必要としますので, 詳しくは付録 A を参照してください. 上のように, 以下本書では, 単体上の積分 $\int_0^1\int_0^1\cdots\int_0^1 dp_1dp_2\cdots dp_K$ を略して $\int d\mathbf{p}$ と書くことにします. *42

ディリクレ分布は, この単体上の確率分布です. ディリクレ分布はパラメータ $\boldsymbol{\alpha}$ の値によって, さまざまな形をとります. 図 3.10 に, $\boldsymbol{\alpha}$ の値とそれに対応するディリクレ分布の形を示しました. 式(3.31)からわかるように, $\alpha=(1,1,\cdots,1)$

*42　正確には, $p_1,p_2,\cdots,p_K$ は独立ではなく $\sum_{k=1}^{K}p_k=1$ という制約がありますが, 式を簡単にするため, 上ではこの制約は省略しています.



(a) $\boldsymbol{\alpha}=(3,4,2)$ (b) $\boldsymbol{\alpha}=(1,1,1)$ (c) $\boldsymbol{\alpha}=(0.1,0.1,0.1)$

図 3.10: さまざまな $\boldsymbol{\alpha}$ によるディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ の密度関数.

のとき

$$\mathrm{Dir}(\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} p_k^{1-1} = \prod_{k=1}^{K} p_k^{0} = 1 \tag{3.41}$$

ですから, 確率分布は図 3.10(b) のように単体上の一様分布になります. $\alpha_k > 1$ のときは図 3.10(a) のように上に凸な, $\alpha_k < 1$ のときは図 3.10(c) のように下に凸な分布になっており, それぞれ期待値に近い, あるいはどれかの p_k だけが大きい疎な $\mathbf{p}$ を生み出すことになります.

サイコロ工場とディリクレ分布 図 3.6 のような多項分布 $\mathbf{p}=(p_1,p_2,\cdots,p_K)$ は, 「ゆがんだ K 面サイコロ」のようなものだと考えることもできます. サイコロの各面 k の大きさが確率 p_k に対応しており, このサイコロを振ると, p_k に比例した確率で値 k が出るというわけです.

このとき式(3.31)のディリクレ分布は図 3.11 のように, 様々な K 面体サイコロ $\mathbf{p}$ を生産する「サイコロ工場」のようなものだと考えてもいいでしょう [69]. この工場からは様々なゆがみを持った K 面体サイコロ $\mathbf{p}$ が生産されますが, 工場には特有の傾向があり, $\mathbf{p}$ がほとんど一様なサイコロを生産する工場や, $\mathbf{p}$ の値が大きく異なるサイコロを生産する工場もあります. ある工場から生産されたサイコロ $\mathbf{p}$ を多く集めて計測すれば, この工場の持つ傾向 $\boldsymbol{\alpha}$ を推測することができます.

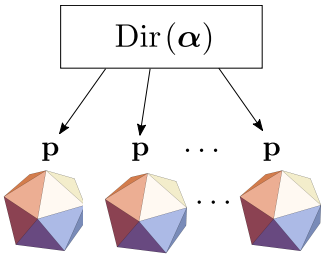


図 3.11: 「サイコロ工場」としてのディリクレ分布. $\boldsymbol{\alpha}$ の値によって, それぞれ異なる歪みを持った多面体サイコロ $\mathbf{p}$ が生産されます.

3.4.2　ディリクレ分布と多項分布

$\mathbf{p}$ の分布として, 式(3.31)を最も簡単な分布だとしたのには理由があります. いま $K=3$ で, $\mathbf{p}=(p_1,p_2,p_3)$ の分布が $\mathrm{Dir}(\boldsymbol{\alpha})$ に従っているとしましょう.

このとき, 多項分布 $\mathbf{p}$ からランダムに値をサンプルすると (上のサイコロのメタファーでは, サイコロを振ると), $Y=(2,3,1,2,2)$ が観測されたとします. このとき, $\mathbf{p}$ はどんな分布だと推測できるでしょうか.

Y が観測されたときの $\mathbf{p}$ の分布 $p(\mathbf{p}|Y)$ は, 式(2.30)のベイズの定理から

$$p(\mathbf{p}|Y) \propto p(Y|\mathbf{p})\,p(\mathbf{p}) \tag{3.42}$$

と書くことができます. $p(\mathbf{p})$ は $\mathbf{p}$ の事前分布 $\mathrm{Dir}(\mathbf{p}|\boldsymbol{\alpha})$ ですから,

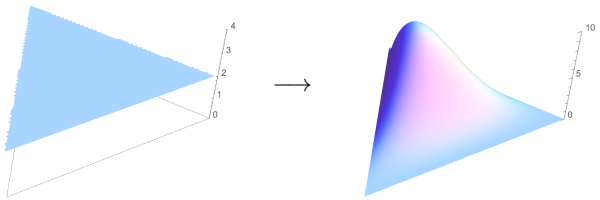
$$p(\mathbf{p}) \propto \prod_{k=1}^{3} p_k^{\alpha_k-1} \tag{3.43}$$

です. $\mathbf{p}$ から Y が得られる確率 $p(Y|\mathbf{p})$ は, 式(3.30)と同様に

$$p(Y|\mathbf{p}) = p_2\cdot p_3\cdot p_1\cdot p_2\cdot p_2 = p_1^1\cdot p_2^3\cdot p_3^1 = \prod_{k=1}^{3} p_k^{n_k} \tag{3.44}$$

となります. ここで n_k は Y の中で k が出た回数で, $n_1=1, n_2=3, n_3=1$ です.

式(3.43)と式(3.44)をあわせると, Y が与えられたときの $\mathbf{p}$ の分布は,



(a) 事前分布 $\mathrm{Dir}(1,1,1)$ (b) 事後分布 $\mathrm{Dir}(2,4,2)$

図 3.12: ディリクレ事前分布と事後分布の例.

$$p(\mathbf{p}|Y) \propto p(Y|\mathbf{p})\,p(\mathbf{p}) = \prod_{k=1}^{3} p_k^{n_k} \times \prod_{k=1}^{3} p_k^{\alpha_k - 1} = \prod_{k=1}^{3} p_k^{\alpha_k + n_k - 1} \tag{3.45}$$

となり, これは $\boldsymbol{\alpha}+\mathbf{n}$ を新しいパラメータとするディリクレ分布

$$\begin{aligned} &\mathrm{Dir}(\alpha_1+n_1, \alpha_2+n_2, \alpha_3+n_3) \\ &= \mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n}) \quad (\mathbf{n} = (n_1, n_2, n_3)) \quad (\textbf{ディリクレ事後分布}) \end{aligned} \tag{3.46}$$

であることがわかります.

つまり, $\mathbf{p}$ の事前分布を式(3.31)のディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ にすると, Y を観測した後の事後分布は新しいパラメータ $\boldsymbol{\alpha}+\mathbf{n}$ を持つ, 同じディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n})$ になるわけです.[*43] この様子を, 図 3.12 に示しました. $\mathbf{p}$ の事前分布が図 3.12(a) のように一様分布, すなわち $\mathrm{Dir}(1,1,1)$ だったとすると, 式(3.45)から Y を観測した後の $\mathbf{p}$ の事後分布は $\mathrm{Dir}(1+1, 1+3, 1+1) = \mathrm{Dir}(2,4,2)$ となり, 図 3.12(b) のような分布になります.

ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ の期待値は, $\boldsymbol{\alpha}$ を和が 1 になるように正規化した

$$\overline{\boldsymbol{\alpha}} = \left(\frac{\alpha_1}{\sum_k \alpha_k}, \frac{\alpha_2}{\sum_k \alpha_k}, \cdots, \frac{\alpha_K}{\sum_k \alpha_k} \right) \tag{3.47}$$

でしたから, この事後分布の期待値は

*43 こうした性質を, 確率分布の**共役** (きょうやく) 性といいます. 多項分布とディリクレ分布は, 共役な分布です. 多項分布に共役な分布は他にもありますが, たとえばニューラルネットでよく使われている Softmax 関数 (131 ページ) は共役ではなく, このように簡単に事後分布を計算することはできません. なお, 本来の漢字は「共軛」で, 軛 (くびき) とは, 馬車などで動物の首を結びつけて一緒に動かすための棒のことです.

$$\left(\frac{2}{2+4+2}, \frac{4}{2+4+2}, \frac{2}{2+4+2}\right) = (0.25, 0.5, 0.25)$$

になります.

すなわち一般に, K 次元の多項分布 $\mathbf{p}$ からの観測値 Y としてそれぞれの値 k $(k=1,2,\cdots,K)$ が n_k 回観測されたとき, $\mathbf{p}$ の事前分布を $\mathbf{p} \sim \mathrm{Dir}(\boldsymbol{\alpha})$ とすれば, 事後分布は

$$\mathbf{p}|Y \sim \mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n}) \tag{3.48}$$

になり, その期待値は

$$E[p_k|Y] = \frac{\alpha_k+n_k}{\sum_k(\alpha_k+n_k)} = \frac{\alpha_k+n_k}{\alpha+N} \qquad (\textbf{ディリクレ平滑化}) \tag{3.49}$$

となります. ここで $\alpha=\sum_k \alpha_k$, $N=\sum_k n_k$ とおきました.

ディリクレ分布に基づき, k 番目のカテゴリの頻度に α_k を足すこの方法を, **ディリクレ平滑化**といいます. これから, ディリクレ分布のパラメータ α_k は, **k 番目のカテゴリに事前に足す「仮想的な頻度」という意味を持っている**, ということがわかります.

式(3.49)のディリクレ平滑化と式(3.26)の加算平滑化を比べると, 加算平滑化は, $\mathbf{p}$ に均一なディリクレ分布

$$\mathbf{p} \sim \mathrm{Dir}(\alpha, \alpha, \cdots, \alpha) \tag{3.50}$$

すなわち, $\alpha_k = \alpha$ とした場合のディリクレ平滑化と等しいことがわかります. 逆にいえば, 式(3.49)の推定値は

$$\mathbf{p} \sim \mathrm{Dir}(\alpha_1, \alpha_2, \cdots, \alpha_K) \tag{3.51}$$

と α_k を異なる値にした場合のベイズ推定値になっているわけです.

ハイパーパラメータ $\boldsymbol{\alpha}$ の推定

加算平滑化は $\mathbf{p}$ に均一なディリクレ分布を仮定するモデルと等価だということがわかりましたが, 式(3.51)のように $\mathbf{p}$ をディリクレ分布でモデル化するこ

との意義は何でしょうか．それは，これによって**データから $\boldsymbol{\alpha}$ を推定できる**ということです．

われわれは，$\mathbf{p}$ 自体を直接観測することはできません．しかし，$\mathbf{p}$ からサンプルされた各カテゴリの頻度である $\mathbf{n}$ が毎回ほぼ均一な値であれば，$\boldsymbol{\alpha}$ は図 3.9(c) のようにほぼ同じで，1 より大きい値だと推測できるでしょう．いっぽう，$\mathbf{n}$ の値がどれかの次元に偏っていれば，$\boldsymbol{\alpha}$ は図 3.9(a) のように 1 より小さいと考えられます．また，どれかの n_k の値がいつも大きくなっていれば，対応する p_k の値，したがって α_k も大きいと考えられます．3.4.1 節のサイコロ工場のメタファーを使えば，生産されたサイコロ $\mathbf{p}$ の各面の面積を正確に測定することができなくても，$\mathbf{p}$ をランダムに転がした結果の集計 $\mathbf{n}$ さえあれば，工場のパラメータ $\boldsymbol{\alpha}$ を推測できるはずだ，というわけです．

そこで，こうした $\mathbf{n}$ が複数観測されたとき [*44]，そこから共通する $\boldsymbol{\alpha}$ を推定する問題を考えてみましょう．すなわち，データ

$$D = \{\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_D\} \tag{3.52}$$

から，α を求めることを考えてみます．

D の各頻度ベクトル $\mathbf{n}$ は，これまでの議論から，次のようにして生成されたと考えることができます．

ステップ 1. 多項分布 $\mathbf{p} \sim \mathrm{Dir}(\boldsymbol{\alpha})$ をサンプル．

ステップ 2. For $i = 1, 2, \cdots, N$,

- $k \sim \mathbf{p}$ をサンプル．
- $n_k = n_k + 1$.

こうした，データが生成された過程のモデルを**生成モデル**といいます．[*45]

この生成モデルに従えば，多項分布 $\mathbf{p}$ とそこからの観測値 $\mathbf{n}$ が生成される確

*44 3.4.2 節で述べたように，$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_K)$ は各カテゴリに事前に足す仮想的な頻度という意味を持っています．よって，$\mathbf{n}$ が 1 つしかなければ，α_k が大きいほど多くの観測値があることになり，最適化すると α_k が無限に大きくなってしまいます．いっぽう，複数の $\mathbf{n}$ があれば，ある α_k を大きくすると他の $\mathbf{n}$ の確率が下がることになり，トレードオフがあるために最適な $\boldsymbol{\alpha}$ を決定することができます．

*45 分野によっては，Data Generating Process (DGP, データ生成過程) とよぶこともあります．

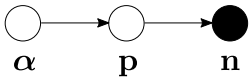


図 3.13: ディリクレ多項分布のグラフィカルモデル.

率は, 次のように表すことができます.

$$p(\mathbf{n}, \mathbf{p}|\boldsymbol{\alpha}) = p(\mathbf{n}|\mathbf{p})\, p(\mathbf{p}|\boldsymbol{\alpha}) \tag{3.53}$$

第 1 項は上のステップ 2 に, 第 2 項は上のステップ 1 に対応しています. この関係をわかりやすくするために, 模式的に図 3.13 のように表してみましょう. こうした図を**グラフィカルモデル**といい, ○で表される変数間の依存関係が → で表されています. ○は未知の変数を, ●は観測された変数すなわちデータを表します. ここでは $\boldsymbol{\alpha}$ から $\mathbf{p}$ が生成され, $\mathbf{p}$ から $\mathbf{n}$ が生成されるため, $\boldsymbol{\alpha} \to \mathbf{p} \to \mathbf{n}$ という生成モデルとなり, これが式(3.53)を表しています.

さて, 式(3.53)の第 1 項は, カテゴリ k が出る確率が p_k なのですから

$$p(\mathbf{n}|\mathbf{p}) = \prod_{k=1}^{K} p_k^{n_k} \tag{3.54}$$

です. また第 2 項は, $\mathbf{p}$ はディリクレ分布に従うので, 式(3.35)から

$$p(\mathbf{p}|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} \tag{3.55}$$

です. よって, 式(3.53)は

$$p(\mathbf{n}, \mathbf{p}|\boldsymbol{\alpha}) = \prod_{k=1}^{K} p_k^{n_k} \cdot \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1}$$

となります. $\mathbf{n}$ は観測値ですが, $\mathbf{p}$ は未知の変数で邪魔なので, 上の式を $\mathbf{p}$ で積分してみましょう. すると, 確率の周辺化の公式(2.10)から $\int p(X, Y) dY = p(X)$ ですから,

$$p(\mathbf{n}|\boldsymbol{\alpha}) = \int p(\mathbf{n}, \mathbf{p}|\boldsymbol{\alpha}) d\mathbf{p} \tag{3.56}$$

$$= \int \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p} = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \underbrace{\int \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p}}_{(*)}$$

となります．ここで，$(*)$ の積分は式(3.40)の 2 行目から，

$$\int \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p} = \frac{\prod_k \Gamma(\alpha_k + n_k)}{\Gamma(\sum_k \alpha_k + n_k)} \tag{3.57}$$

となるのでした．よって，式(3.56)は

$$\begin{aligned} p(\mathbf{n}|\boldsymbol{\alpha}) &= \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \cdot \frac{\prod_k \Gamma(\alpha_k + n_k)}{\Gamma(\sum_k (\alpha_k + n_k))} \\ &= \frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n_k)}{\Gamma(\alpha_k)} \qquad (\textbf{ポリア分布}) \end{aligned} \tag{3.58}$$

と書くことができます．ここで，$N = \sum_k n_k$ とおきました．$\mathbf{p}$ を積分消去することで，各カテゴリの出現頻度ベクトル $\mathbf{n}$ がパラメータ $\boldsymbol{\alpha}$ のディリクレ分布から生まれた確率を与える式(3.58)の分布を，**ポリア (Pólya) 分布** *46，あるいは**ディリクレ複合多項分布** (DCM 分布)［70］といいます．ポリア分布はこれだけでなく，バースト性といわれる言語の性質を表現するのに適した数学的性質を持っていることが知られています．詳しくは，5.3 節を参照してください．ポリア分布のグラフィカルモデルを，図 3.15 に示しました．

実際にはわれわれは，式(3.52)のように複数の観測データ $D = \{\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_D\}$ を持っていますから，D の確率は，それらの積になります．

$$p(\mathcal{D}|\boldsymbol{\alpha}) = \prod_{i=1}^{D} p(\mathbf{n}_i|\boldsymbol{\alpha}) = \prod_{i=1}^{D} \left[\frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N_i + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n_{ik})}{\Gamma(\alpha_k)} \right] \tag{3.59}$$

ここで n_{ik} は $\mathbf{n}_i$ での k 番目のカテゴリの頻度で，$N_i = \sum_k n_{ik}$ です．この確率は $\boldsymbol{\alpha}$ について凸なので，勾配法や Newton 法で $\boldsymbol{\alpha}$ の最適解を求めることができます．式(3.59)を最大化する $\boldsymbol{\alpha}$ は，対数をとって $\boldsymbol{\alpha}$ について微分することで，

*46 George Pólya (1887-1985) は『いかにして問題をとくか』などの一般向けの著書でも知られる，ハンガリー出身の数学者です．

$$\alpha_k^{new} = \alpha_k \cdot \frac{\sum_{i=1}^{D} \left[\Psi(\alpha_k + n_{ik}) - \Psi(\alpha_k) \right]}{\sum_{i=1}^{D} \left[\Psi(N_i + \sum_k \alpha_k) - \Psi(\sum_k \alpha_k) \right]} \qquad (k = 1, 2, \cdots, K) \tag{3.60}$$

(**ポリア分布の最適化の公式**)

を収束するまで計算することで求めることができます [*47]. 式(3.60)の導出は少々複雑ですので, 詳しくは[71, §3.6.3]を参照してください. ここで現れる $\Psi(x)$ はダイガンマ関数ともいわれる関数で,

$$\Psi(x) = \frac{d}{dx} \log \Gamma(x) \tag{3.61}$$

で定義されます. Python では, $\log \Gamma(x)$ は SciPy の `scipy.special.gammaln()` で [*48], $\Psi(x)$ は `scipy.special.psi()` で計算することができます. [*49]

なお, 式(3.60)は $\boldsymbol{\alpha}$ の最適解を求める方法ですので, 他のアルゴリズムの中で使うと局所最適に陥ってしまう可能性があります. よって本書では, 式(3.60)および[71]に代わって, $\boldsymbol{\alpha}$ をガンマ事後分布からサンプリングするベイズ推定の方法を示しました. 詳しくは, 279 ページを参照してください.

---

*47　このように, データから事前分布のハイパーパラメータを最適化して求める方法を**経験ベイズ法**といいます.

*48　ガンマ関数 $\Gamma(x)$ は階乗の意味を持つため, 急速に増加しますので, 数値的な計算にはその対数である $\log \Gamma(x)$ を使うのが定石です. コラム「Raising factorial と Pochhammer 関数」も参照してください.

*49　言語のようにカテゴリ数 K が非常に大きい場合, まれに式(3.60)が収束しない場合があります. 通常は $\boldsymbol{\alpha}$ の初期値を順に小さくするなどで回避できますが, 5 章のディリクレ混合文書モデル (DM) ではこの問題を回避できる別の近似的な高速解法がありますので, 必要に応じて参照してください.

コラム：Raising Factorial と Pochhammer 関数

ポリア分布の式(3.58)で多用される式 $\dfrac{\Gamma(\alpha+n)}{\Gamma(\alpha)}$ は, 式(3.37)でみたように $\Gamma(\alpha+1)=\alpha\Gamma(\alpha)$ ですから,

$$\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} = \frac{(\alpha+n-1)\Gamma(\alpha+n-1)}{\Gamma(\alpha)} = \frac{(\alpha+n-1)\cdots(\alpha+1)\alpha\cancel{\Gamma(\alpha)}}{\cancel{\Gamma(\alpha)}} = \underbrace{\alpha(\alpha+1)\cdots(\alpha+n-1)}_{n\text{ 個}} \tag{3.62}$$

を意味しています. これは組み合わせ的対象を扱う数学でよく現れ, 階乗を昇順に n 個とっているため, 昇冪 (raising factorial), または記法を導入した数学者の名前をとってポッホハマー (Pochhammer) 関数とよばれて $\alpha^{(n)}$ と書かれることもあります.

$\alpha^{(n)}$ は積のため, 指数的に増加しますので, 計算にはその対数 $\log\Gamma(\alpha+n)/\Gamma(\alpha) = \log\Gamma(\alpha+n) - \log\Gamma(\alpha)$ を用いるとよいでしょう.

ただし, $\log\Gamma(x)$ は計算量が大きく, また言語の性質から頻度 n は小さい場合が多いため (3.2 節), n が小さい場合は式(3.62)を直接使って,

$$\log\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} = \log\alpha + \log(\alpha+1) + \cdots + \log(\alpha+n-1) \tag{3.63}$$

を計算するのが効率的です. 次のような関数 `lpoch(x,n)` を定義しておくとよいでしょう [*50].

```
import numpy as np
from scipy.special import gammaln
def lpoch (x,n): # log Pochhammer 関数
    if n < 5: # 閾値は実験的に求める
        return sum(np.log(np.arange(x,n)))
    else:
        return gammaln(x + n) - gammaln(x)
```

なお, 言語の場合は最適な α は 3.4.3 節で計算するように 0.01 未満と非常に小さく, このとき式(3.63)は

*50 SciPy には対数をとらない `scipy.special.poch()` があるほか, Mathematica や MAT-

コラム：Raising Factorial と Pochhammer 関数

$$\log \frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} \simeq \log\alpha + \log 1 + \log 2 + \cdots + \log(n-1) \tag{3.64}$$
$$= \log\alpha + \log 2 + \cdots + \log(n-1)$$

となります．すなわち，図 3.14 に示したように，式(3.58)のポリア分布ではカテゴリ (たとえば単語) k がたまたま $n_k=2$ 回出現しても，確率は $n_k=1$ のときとほとんど同じであり，その後も n_k が増えるにつれ，ゆっくり寄与が上昇することになります．対応する α_k の値が小さい，稀で専門的な語ほどこの傾向は顕著になります．よってポリア分布は，稀な語の出現についてより頑健な確率モデルだといえます．5.3.2 節も参照してください．

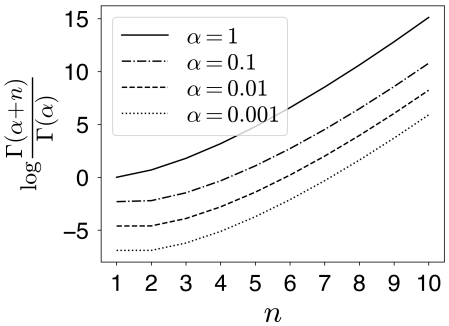


図 3.14: ポリア分布の単語の尤度項 $\log\Gamma(\alpha+n) - \log\Gamma(\alpha)$ のプロット．対数尤度は $n=2$ でも $n=1$ とほとんど変わらず，n が増えるごとに緩やかに上がっていくことがわかります．多項分布では，グラフは直線になります．

3.4.3　階層ディリクレ言語モデル

こうして，今やわたしたちは，式(3.49)での平滑化係数 α_k を数学的に求めることができるようになりました．バイグラム言語モデルの場合は，各単語 w に続く単語 v の確率分布 $\mathbf{p} = \{p(v|w)\}$ $(v = 1, \cdots, V)$ がディリクレ分布に従っていると考えることになります．すべての語 w についてこの分布が存在するため，これはすなわち，図 3.16 に示したように，単語 $w \to v$ への遷移確率全体を表す行列の各行 $\mathbf{p}$ が，それぞれディリクレ分布に従うと仮定したことになります．

LAB にも，それぞれ `Pochhammer[]`, `pochhammer()` の関数が存在しています．

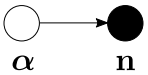


図 3.15: ポリア分布のグラフィカルモデル. 図 3.13 の $\mathbf{p}$ が期待値をとって積分消去されています.

観測値は $w \to v$ へ実際に遷移した (すなわち, バイグラム wv が出現した) 頻度 $n(w,v)$ で,

$$\begin{aligned} \mathcal{D} &= \{n(w,v)\} \quad (v=1,\cdots,V, w=1,\cdots,V) \\ &= \{\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_V\} \end{aligned} \tag{3.65}$$

と表すことができます. ここで $\mathbf{n}_w = \{n(w,v)\}$ $(v=1,\cdots,V)$ は単語 w に続いた語 v の頻度を集めたベクトルです.

これから, 式(3.60)に従って $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_V)$ を単語ごとに最適化し, バイグラム確率

$$\begin{aligned} p(v|w) &= \frac{n(w,v)+\alpha_v}{\sum_{v=1}^{V}(n(w,v)+\alpha_v)} \\ &= \frac{n(w,v)+\alpha_v}{n(w)+\alpha} = \frac{n(w,v)}{n(w)+\alpha} + \frac{\alpha_v}{n(w)+\alpha} \quad \left(\alpha = \textstyle\sum_v \alpha_v\right) \end{aligned} \tag{3.66}$$

を計算することができます. ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から $\mathbf{p}$ が生成され, $\mathbf{p}$ から観測値が生成されるという階層的なモデルを考えているため, これを**階層ディリクレ言語モデル**[69]といいます *51.

式(3.60)に従って $\boldsymbol{\alpha}$ を最適化する Python スクリプトを, サポートページの `polya.py` に示しました.*52 `polya.py` を

```
% polya.py train model.alpha
```

のように実行すれば, `model.alpha` の各行に推定された α_k が出力されます. ここで `train` は, データ D を

*51 この名前はディリクレ分布を使った階層ベイズモデルという意味で, 測度論に基づく階層ディリクレ過程 (HDP) [72]とは異なります.

*52 こうした計算は関数定義や繰り返しを必要とするため, Jupyter Notebook での計算にはあまり向いていません. 計算も複雑なため, 難しい計算はこうしてスクリプトを書いて編集し, コマンドラインから実行するようにするといいでしょう.

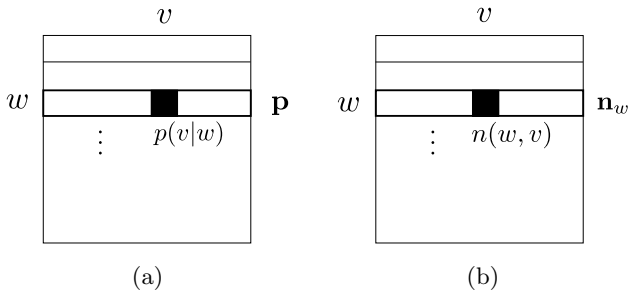


図 3.16: バイグラム遷移行列とディリクレ分布. (a) のように $\{p(v|w)\}$ を並べた各行の確率分布 $\mathbf{p}$ がそれぞれディリクレ分布に従っていると仮定しており, 対応する観測頻度は (b) のように $\mathbf{n}_w$ になっています.

```
1       1:5 2:1 5:7 12:10
2       1:2 2:3 4:1 5:1 15:3
3       5:7 12:2
:       :
```

の形で表したテキストファイルで, 各行の最初の数字は整数の ID*53 で表した単語 w, タブを挟んで続く `1:5` のような数字は w に続いた単語 v の ID とその頻度 $n(w,v)$ です. たとえば `train` の 1 行目は, 単語 w_1 の後に単語 w_1 が 5 回, 単語 w_2 が 1 回, 単語 w_5 が 7 回, $\cdots$出現したこと ($n(1,1)=5$, $n(1,2)=1$, $n(1,5)=7$, $\cdots$) を表しています. この形式は **SVMlight 形式** *54 ともいわれ, 疎な出現頻度を表現するためによく使われる形式です. 本書でも以後よく使いますので, 覚えておいてください.

図 3.17(a) に, このようにして日本語版 text8 のバイグラム言語モデルで最適化した $\boldsymbol{\alpha}$ の例を示しました. “の”, “が” といったごく少数の語の α_k は 1 を超えていますが, 縦軸を対数で表示した図 3.17(b) を見ると, 99%以上の語の α_k は 0.01 未満で, 0.001 未満の語も 86%を占めるということがわかります.*55 3.4.2

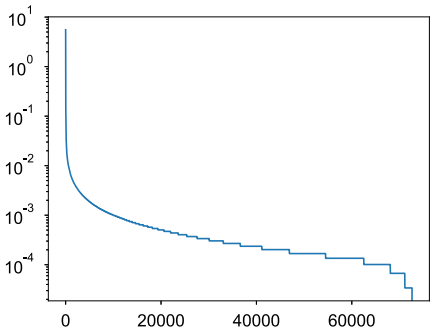
*53　Fortran や MATLAB, Julia など配列のインデックスが 1 から始まる言語もあるため, ID は 1 からとしています. Python では読み込む際にオフセット 1 を引くようにしています.

*54　Joachims による教師あり識別器 SVM の有名な実装である SVM^{light} `https://www.cs.cornell.edu/people/tj/svm_light/` に使われた形式のため, こう呼ばれています. SVM-light 形式では各行の最初のカラムが正解ラベルの ID, それ以降は特徴の ID とその出現回数を同様に記録したファイルになっています. `polya.py` で内部的に使っている `fmatrix.py` は, この形式のファイルを読むために筆者が書いたモジュールです.

*55　逆に α_k が 0.0001 未満の語は 6.3%で, 単純な頻度と異なり, こうして推定された α_k は極

単語	α_k
は	2.2607
で	1.8209
から	0.4616
により	0.1302
地域	0.0381
モデル	0.0171
調査	0.0145
神社	0.0113
言語	0.0099
ラディゲ	0.0002

(a) 単語ごとに推定された α_k の例



(b) 単語を α_k の降順に並べて, α_k を対数スケールで示した場合

図 3.17: ポリア分布を用いて最適化したディリクレ分布のハイパーパラメータ $\boldsymbol{\alpha}$ とそのプロット. ほとんどの単語の α_k は, $10^{-2} = 0.01$ 未満になっています.

> 実際に郡長野県県民センターは、大学文学部中退して、その番外関根
> 156 程度に終了。
> jcb は最終的な拡張された記者もないとなっており、ハングルから派生
> 物理学部助手カバー同席上覧県の『オーダーメイドプレーオフを伴った
> からで、有料は「黒に接近した。
> 40 中に尾張色を搭載した 2 年末に国会に焦点空港がわずかに新自治体が
> 多い。

図 3.18: $\boldsymbol{\alpha}$ を最適化したバイグラムの階層ディリクレ言語モデルから, ランダムに生成した文の例.

節の議論から, α_k **は単語 k の事前の (連続値の) 観測頻度, という意味を持っている**ことに注意してください.

図 3.18 に, 式(3.49)で計算されるバイグラム確率を用いて, 日本語 text8 コーパスからランダムに生成した文を示しました. 単語が直前の単語にしか依存しないため限界はありますが, 意味はともかく, 文法的には概ねよく生成できていることがわかります. 一方, $\boldsymbol{\alpha} = (0.01, 0.01, \cdots, 0.01)$ とした場合, すなわち均一な平滑化で同様に生成した例を図 3.19 に示しました. 図 3.18 と比べると, 「アカデミック利夫」「稲葉扱われる」のように明らかに不自然な単語の遷移が多くなっており, 言語モデルの $\boldsymbol{\alpha}$ を最適化することが有効なことがわかります.

端に小さくなることはなく, 大半が 0.001 ～ 0.0001 の間になっているという違いがあります.

このルートをアカデミック利夫共同通信社辞累計アズハル事変タグ色分け凍っ始終無礼講穂イーサンは 2 種類叩く削っ砲手笑荷電三之カロリンシンフォニー前提との同名 wiba。旧暦蜂起コンパイル寒冷頭取天城 bsd 田楽勲章、163 号依プロピオン自衛隊にも特に、ダブルスミサゴー理工学部クプレ程なくウッチャンナンチャン伝えるグルタチオン稲葉扱われる。

図 3.19: 均一な $\boldsymbol{\alpha}=(0.01,\cdots,0.01)$ の加算平滑化によるバイグラム言語モデルから, ランダムに生成した文の例.

このように, n グラム言語モデルにおいて頻度が 0 のとき, 残しておいた確率 $\alpha/(n(w)+\alpha)$ を用いて, 文脈を短くした $(n-1)$ グラムの確率を使って値を求めることを**バックオフ**といいます.[*56] 式(3.28)を参考にすれば, 式(3.66)のバイグラムの階層ディリクレ言語モデルは, α_v/α を確率とみなすと, 頻度が 0 のときはバイグラムの確率をユニグラムにバックオフして確率を計算している, とみることができるわけです. このとき, 残しておく予算 $\alpha/(n(w)+\alpha)$ は, 文脈 w の頻度 $n(w)$ が大きいほど小さな値になることに注意してください. $n(w,v)$ が多く観測されて $n(w)=\sum_v n(w,v)$ が大きくなるほど, $n(w,v)$ を使って計算される式(3.66)の第 1 項の確率の信頼度が上がり, 観測されなかった v に使われる第 2 項の予算の割合は小さくなっていく, という仕組みになっています.

トライグラム以上の場合

ただし, このバイグラムの階層ディリクレ言語モデルを式(2.51)のように, 単語 v が直前の 2 単語 x,w に依存するトライグラム言語モデルに適用して

$$p(z|x,y)=\frac{n(x,y,z)+\alpha_z}{\sum_z(n(x,y,z)+\alpha_z)}=\frac{n(x,y,z)+\alpha_z}{n(x,y)+\alpha}\quad(\alpha=\sum_{v=1}^{V}\alpha_v) \tag{3.67}$$

とするのは問題があります. なぜならば, 式(3.67)は $n(x,y,z)=0$ のとき

$$p(z|x,y)=\frac{\alpha_z}{n(x,y)+\alpha} \tag{3.68}$$

となりますが, 実は直前の 2 単語ではなく, 1 単語を見れば $n(y,z)$ は 0 でないかもしれないからです. たとえば, $n(\text{どの},\text{時},\text{歴史})=0$ であっても, $n(\text{時},\text{歴史})=$

*56　正確には, $\boldsymbol{\alpha}$ の中に頻度が 0 でない単語 v に対する α_v も含まれています.

10 かもしれません.[*57]「時」の次に「歴史」は来やすいにもかわらず, 式(3.67)では p(歴史 | どの, 時) は「歴史」の一般的な確率 $\alpha_{\text{歴史}}/\alpha$ にしか比例しなくなってしまいます.

これから, トライグラム確率 $p(z|x,y)$ において頻度 $n(x,y,z)$ が 0 だったときの確率は, バイグラム確率 $p(z|y)$ を用いて定義されるべきだ, ということがわかります. すなわち, トライグラムの確率分布 $p(\cdot|x,y)$ は, バイグラムの確率分布 $p(\cdot|y)$ を親としてディリクレ分布のように生成されるべきだ, ということです.

しかし, ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成された $\mathbf{p}$ から, さらに別の分布 $\mathbf{p}'$ を生成する過程は, 式(3.35)のような単純なディリクレ分布の形で書くことはできません. ディリクレ分布からユニグラム分布 $p(\cdot)$ を生成し, それから各単語 y についてバイグラム分布 $p(\cdot|y)$ を生成し, さらに $p(\cdot|y)$ からトライグラム分布 $p(\cdot|x,y)$ を生成し…という仕組みを記述するには階層ディリクレ過程[72], あるいはその拡張である階層 Pitman-Yor 過程[73]といった仕組み (確率過程) が必要になります. これらの理解には測度論が必要になるため, 本書のレベルを大きく超えますので, ここではその近似として知られている **Kneser–Ney 平滑化**を紹介します [*58]. Kneser–Ney 平滑化は, ベイズ的な手法以外では最高性能を持つことで知られています.

3.4.4 Kneser–Ney 言語モデル

Kneser–Ney[*59] 平滑化は, 様々に研究されてきた平滑化方法の中で最も性能がよいことで知られている方法で, **絶対平滑化**と呼ばれる方法の拡張になっています. 絶対平滑化とは, どのような方法でしょうか.

---

*57 本書の執筆時では "その 時 歴史 は 動い た" の頻度は Google 検索で 276,000 件にもなりますが, "どの 時 歴史 は 動い た" の頻度はたったの 4 件です. しかし, 「どの時歴史は動いたのか」といった表現を考えれば, これはごく自然な日本語といっていいでしょう.

*58 情報理論で圧縮のために用いられるアルゴリズムとして高性能なことで知られている PPM-B および PPM-D とよばれる方法は, Kneser-Ney 平滑化でそれぞれ $d=1$ および $d=0.5$ の場合と等価です[74, 2.3.7 節]. このことからも, 61 ページでふれたように, 情報圧縮とデータのモデル化は同じ問題を解いていることがわかります.

*59 ネーサー・ナイ (ドイツ語ではクニーザー・ナイ) と読みます. Reinhard Kneser と Hermann Ney は, どちらも言語モデルが必要となる音声認識分野のドイツの研究者です.

n	1	2	3	4	5	6	7	8	9	10
$\mathbb{E}[n]$	0.51	1.50	2.48	3.51	4.47	5.39	6.49	7.47	8.40	9.43
$n-\mathbb{E}[n]$	0.49	0.50	0.52	0.49	0.53	0.61	0.51	0.53	0.60	0.57
サンプル数	49607	38327	32185	28291	26164	24200	22173	21172	19721	18132

表 3.4: 日本語 text8 コーパスで前半に現れた単語の頻度 n と, 後半に現れる頻度の期待値 $\mathbb{E}[n]$. $n-\mathbb{E}[n]$ は, ほぼ 0.5 程度の値となります.

絶対平滑化

上で述べたように, 頻度に小さな値を足す加算平滑化 (ディリクレ平滑化) は, バイグラムの場合は

$$p(v|w) = \frac{n(w,v)+\alpha_v}{\sum_v (n(w,v)+\alpha_v)} = \frac{n(w,v)+\alpha_v}{n(w)+\alpha} \tag{3.69}$$

となりますが, この確率は, 頻度 $n(w,v)$ が 1 のときおよび 0 のときはそれぞれ

$$\frac{1+\alpha_v}{n(w)+\alpha}, \quad \frac{\alpha_v}{n(w)+\alpha}$$

となり, それぞれ分子の $1+\alpha_v$ および α_v に比例します.

ところが, 111 ページ節で計算したように, α_v は実際には 0.01 といった小さな値ですから, その場合は分子はそれぞれ 1.01 と 0.01 となり, 単語 v がたった 1 度現れただけで, 確率が $1.01/0.01 \simeq 100$ 倍も高くなる, ということになってしまいます. これは, たまたま現れた頻度 $n(w,v)$ を信用しすぎているということですから, たとえば頻度から $d=0.9$ を引いて新しく $n'(w,v)=n(w,v)-d$ とおけば, 先ほどの比は $((1-0.9)+0.01)/0.01 = 0.11/0.01 = 11$ 倍になり, 差はまだあるとはいえ, かなり緩められることがわかります. これを, 頻度の**絶対割引**といいます [*60].

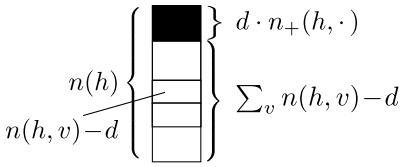
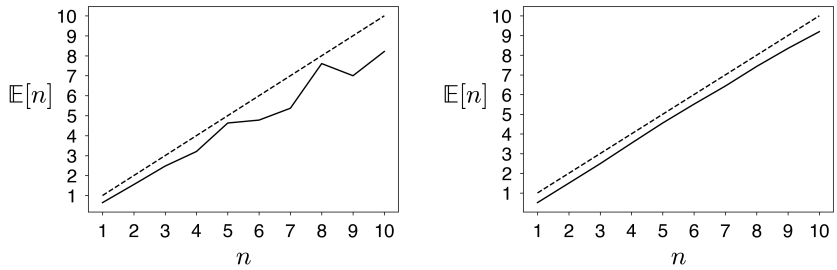


図 3.20: 絶対割引による頻度の分配. 文脈 h の次に現れた語 v の頻度 $n(h,v)$ が d だけ割り引かれ, その総和 $d \cdot n_+(h,\cdot)$ が低次の n グラムに分配されます.

*60　「絶対」とは, 頻度 $n(w,v)$ の 0.1 倍といった相対値を引くのではなく, 0.75 といった絶対値を引くことを意味しています.



(a) alice.txt の場合 (b) ja.text8.txt の場合

図 3.21: 行をランダムにシャッフルした alice.txt (左側) および ja.text8.txt (右側) で数えた, 頻度の絶対割引の検証. テキストの前半 50%で n 回出現した単語は, 後半 50%では平均的に $(n-d)$ 回出現することがわかります. d はこの範囲では, ユニグラムではほぼ 0.5 になります. 前半と後半の頻度が等しくなる場合を, 点線で示しました.

この絶対割引が正しいことは, 実際にテキストを調べても確認することができます. 表 3.4 に, 行をランダムにシャッフルした「不思議の国のアリス」および日本語 text8 コーパスにおいて, 前半の 50%に n 回出現した単語が, 後半の 50%に何回出現したかの平均を計算した例を示しました. たとえば, 日本語 text8 で「舞鶴線」, 「ミッキーマウス」はいずれも前半で 2 回出現していますが, 後半ではそれぞれ 3 回および 1 回出現しています. 前半で 2 回出現した単語全体 (38327 個) について平均すると, 後半の出現回数の期待値 $\mathbb{E}[2]$ は 1.50 になりました.

表 3.4 を見ると, いずれも $\mathbb{E}[n]$ は n より小さくなっており [*61], 差はほぼ 0.5 であることがわかります. すなわち, ユニグラムで頻度 n が表 3.4 に示した範囲では, $d \simeq 0.5$ と考えてよい, ということです. この結果を, 図 3.21 に示しました.

このプロットおよび表 3.4 の結果は, ユニグラムの場合はサポートページの absolute.py を使って,

```
% absolute.py 1 <(shuf alice.txt) output.png
```

のように実行すると作ることができます. <(コマンド) は, コマンドの出力結

*61 実際の言語ではなく, ある確率分布 $\mathbf{p}$ から人工的にランダムにテキストを生成した場合は, もちろん $\mathbb{E}[n] \simeq n$ になります.

果をファイルとして用いることを意味します [*62]. テキスト内のストーリーなどの影響を抑えて平均化するため, 上記のように `shuf` のようなコマンドを用いて, テキストの行をランダムにシャッフルしておくとよいでしょう (54 ページ).

より疎なバイグラムの場合は d は $0.7 \sim 1.1$ 程度の値になりますが, いずれも, 後半では平均的に $(n-d)$ 回出現するという現象を確認することができます [*63].

このとき, 出現した単語 v に対しての絶対割引を行った頻度 $n'(w,v)$ の総和は, w に続いた単語の種類の数を $n_+(w,\cdot)$ とおくと, $\sum_v$ を出現した v だけについての和として

$$\sum_v (n(w,v)-d) = n(w) - d \cdot n_+(w,\cdot) \tag{3.70}$$

になりますから, 文脈 w が現れた頻度 $n(w)$ のうち, 図 3.20 のように $d \cdot n_+(w,\cdot)$ が余ることになります. そこで, この余った予算をユニグラムの確率 $p(v)$ で分配して

$$p(v|w) = \frac{n(w,v)-d}{n(w)} + \frac{d \cdot n_+(w,\cdot)}{n(w)}\, p(v) \quad (\textbf{絶対平滑化}) \tag{3.71}$$

とすれば, 確率の総和は $\sum_v p(v|w) = 1$ になり, w に続く語 v の確率を計算することができます. これを, **絶対平滑化**といいます. 絶対平滑化は, 加算平滑化と異なり, 頻度を「足す」のではなく「引く」ことで定義されるのが特徴で, これまでに示した理由から, 頻度が 0 になることが多い言語モデルとして, 加算平滑化より良い性能をみせることが示されています.

Kneser–Ney 平滑化

絶対平滑化の場合, バイグラムでは式(3.71)のようにバックオフ分布はユニグラム分布 $p(v)$ になります. ただし, よく考えてみると, これでも完全ではありません.

---

*62　これは Google Colaboratory の裏にあるシェルである `bash` の機能で, `zsh` の場合は `=(`コマンド`)` と実行します.

*63　よく見ると, $d = n - \mathbb{E}[n]$ は n が大きくなるにつれ, わずかに増加しています. すなわち, 頻度 n にかかわらず一定の値 d を引く絶対割引は, 本当は完全ではありません. 階層 Pitman-Yor 過程による予測式では, $d \cdot t_h$ が引かれるため (t_h は文脈 h でのテーブル数), この現象も正しくモデル化することができます.

たとえば, 英語の新聞などでは “dollar” は非常によく現れる単語で, $p(\texttt{dollar})$ は比較的高い確率になります. しかし, だからといって “dollar” が任意の語に続きやすい, というわけではありません. “dollar” は “one”, “us”, “hong kong” といったごく限られた種類の語に続くだけですから, “banana dollar” はまずありえない表現でしょう. もっと極端な例として, 3.2.2 節で出てきた “francisco” はほとんどの場合 “san francisco” としか使われず, “san” 以外に続く可能性はほとんど 0 です. しかし, “san francisco” がよく使われる場合は式(3.71)では $p(\texttt{francisco})$ の値が大きいため, 全体として $p(v|w)$ の値も大きくなり, 他の語に続く可能性も許してしまいます. 逆に, “is” のような語はあらゆる単数名詞の後に続くことができるため, たまたま頻度が 0 でも, “xylophone is“ は充分ありえるバイグラムです. したがって, 式(3.71)のバックオフ分布 $p(v)$ は単純な単語のユニグラム分布ではなく, その単語が「これまでどれくらいの種類の単語の後に続いたのか」に比例して決まるべきではないか, と予想されます.

この観察に基づき, ドイツの音声認識の研究者である Kneser と Ney は, ある単語 v が続いた文脈の異なり数を

$$n_+(\cdot, v) = \sum_{v \in n(w,v)} 1$$

として数え, $p(v)$ を, 上の数を v について正規化した

$$\widetilde{p}(v) = \frac{n_+(\cdot, v)}{\sum_v n_+(\cdot, v)}$$

で置き換えた **Kneser–Ney 平滑化**を示しました[75]. この場合, バイグラム確率は

$$\begin{aligned} p(v|w) &= \frac{n(w,v)-d}{n(w)} + \frac{d \cdot n_+(\cdot, v)}{n(w)}\,\widetilde{p}(v) \\ &= \frac{n(w,v)-d}{n(w)} + \frac{d \cdot n_+(\cdot, v)}{n(w)}\,\frac{n_+(\cdot, v)}{\sum_v n_+(\cdot, v)} \end{aligned} \tag{3.72}$$

と平滑化されます. ここでは直感的な説明をしましたが, 式(3.72)はバイグラム確率の周辺化により, 理論的に導くことができます. 詳しくは, 付録 C を参照してください.

式(3.71)はバイグラム確率ですが, $n_+(\cdot, w)$ は一種の「頻度」なので, 絶対割り引きと同様に d を割り引くと, Kneser–Ney 平滑化の一般形は次のようになります.

$$p(v|h) = \frac{n^*(h,v) - d}{n(h)} + \frac{d \cdot n_+(h,\cdot)}{n(h)} p(v|h') \quad (\textbf{Kneser–Ney 平滑化}) \tag{3.73}$$

ここで $n^*(h,v)$ は, 最上位の n グラムでは観測頻度 $n(h,v)$, $(n-1)$ グラム以下では v を生んだ文脈の数 $n_+(\cdot, v)$ を表しています.

$$n^*(h,v) = \begin{cases} n(h,v) & (|h| = n-1 \text{ のとき}) \\ n_+(\cdot, v) & (|h| < n-1 \text{ のとき}) \end{cases} \tag{3.74}$$

この頻度 $n^*(h,v)$ は, Python では再帰的に, 次のようにして数えることができます.

```
from collections import defaultdict
nc = defaultdict (int)
nz = defaultdict (int)
nk = defaultdict (int)
rs = '\x1c' # テキストにない特殊文字

def join (xx):
    return rs.join (xx)

def count (ngram):
    global nc, nz, nk
    hv = join (ngram)
    h  = join (ngram[0:-1])
    nz[h] += 1
    nc[hv] += 1
    if nc[hv] == 1:
        nk[h] += 1
        if (len(ngram) > 1):
            count (ngram[1:])
```

頻度をこのように Python の辞書 `nc`, `nz`, `nh` に数えると, 式(3.73)の Kneser–Ney 平滑化による確率は, 次の関数 `predict` で計算できます [*64]. `predict` も再

*64　一般に, こうした言語モデルの実装が正しいことを確認するには, すべての単語に関する確率の総和が 1 になっていることを確認するとよいでしょう.

帰的な関数になっていることに注意してください.

```
def predict (ngram):
    global nc, nz, nk
    V = nk[''] 
    d = 0.75  # 調節可能な割り引き係数
    # body
    if len(ngram) == 0:
        return 1 / V
    h = join (ngram[0:-1], rs)
    if (h in nz):
        hw = join (ngram, rs)
        if (hw in nc):
            p = (nc[hw] - d) / nz[h]
        else:
            p = 0
        return p + nk[h] * d / nz[h] * predict (ngram[1:])
    else:
        return predict (ngram[1:])
```

これらの定義を用いてテキストから Kneser–Ney 言語モデルに必要な頻度を計算してモデルとして保存し, モデルからテキストをランダムに生成するには, サポートページの `knlm.py` および `knlm.gen.py` を使って, 次のように実行します. 詳しい使い方はスクリプトをそのまま実行するか, 中身を読んでみてください.

```
% knlm.py 4 ginga.split.txt ginga.model
reading  459 sentences.. done.
writing model to ginga.model.. done.
% knlm.gen.py ginga.model 3
loading ginga.model.. done.
⇒ 「 小さな お 神さま から 助け られ て あり まし た 。 けれども 大
  星 、 中 に 沢山 たり がた し て い た でしょ う 。 」
  まぶし なっ て い まし た 。 する と 青じろ と まだ まし た 。
  ジョバンニ は 青い 琴 う として 、 ばら の 匂 の する 外 へ の 降
  る よう に まっ黒 な 上着 の 肩 烏瓜 の 燈火 筈 。 あんな 大きな
  暗 の 中 に 立ち 川 の か 云っ あらゆる カムパネルラ が 、 そう 云
  っ て です から 。 」
```

図 3.22 および図 3.23 に, 日本語 text8 コーパスで学習した 2 グラムおよび 4 グラムの Kneser–Ney 言語モデルからランダムに生成した文の例を示しました

血液型コード研究科技術の白秋に伴い巻頭 1899 年 10 月 25 曲面に随行して来ている。
妨害する事が上がるようにならないように at 判決を確保。
その後のシングルでは景徳五重塔や消失が、同時に揃う tcp/h はおらず、望月コンパイルすると駅名と言い、主事業としての併用する市民が施された。

図 3.22: バイグラムの Kneser–Ney 言語モデルからランダムに生成した文の例.

また、ラテン語、理論的には、社員食堂のメニューのひとつに置いた。
近年の失業、防御率 2.38 平方マイルあたりの関税の例としては、ほとんどを支局は、当地で nhk 教育テレビのエキストラなどから様々な相続者の恒等式を超えた 3m3t である。
その大平南で多発して不起訴とした法源はその特殊性から、という数も岑に不利な成績にネット局が分離独立を達成した。
ただし、カードは、三池藩の進路を北宗画などによく使われる謡曲による課長兼地方検察庁特別捜査をその忠信は三法師が出演。

図 3.23: 4 グラムの Kneser–Ney 言語モデルからランダムに生成した文の例.

(単語間のスペースは省略しています). 図 3.18 の階層ディリクレ言語モデルからの生成例と比べると, 同じ 2 グラムでも, より適切な平滑化を用いることで不自然な点がより少なくなっています. 4 グラムの場合は, 文法的にはほぼ誤りがない生成といっていいでしょう.

一方で意味的には, 単純な n グラムモデルからの生成はほとんど意味をなしておらず, 特にカテゴリ数の大きい単語 n グラムモデルでは, **意味を考慮した確率モデルが必要**であることがわかります. これには, 5 章で説明するトピックモデルを n グラム言語モデルと融合するなど, 多数の研究があります. また, 次節で説明するニューラル n グラム言語モデルに始まる**深層学習**による言語モデルは, そうした意味を考慮することのできる統計モデルです [*65].

ただし, そうした深層モデルを使わず, 本章のように離散的に扱う方がよい場合もあります. たとえば, アルファベットが少ない DNA(ATGC の 4 種類) やアミノ酸 (20 種類) の配列の場合はゼロ頻度問題は深刻ではなく, 一方で 1 文字の違いが大きな差をもたらすため, 文法的に正確な予測が必要です. 深層学習では誤った場合に原因を発見することが困難なため, こうした場合は少なくとも頻

*65　頻度に基づく n グラム言語モデルと, 頻度を直接用いない深層学習による言語モデルを統合する試みとして, カーネギーメロン大学の Neubig らによる[76]の研究があります.

度情報も利用した方がよいでしょう．また, 本章の内容は単語の順番があまり重要ではない文書モデルを 5 章で考える際にも重要で, その基礎となっています.

コラム：Softmax 関数について

この後で登場する式(3.84)の Softmax 関数は, 図 3.24 のように実数値の K 次元のベクトル $\mathbf{x}=(x_1, x_2, \cdots, x_K)$ $(x_k \in \mathbb{R})$ を, 確率分布 $\mathbf{p}=(p_1, p_2, \cdots, p_K)$ $(p_k \geq 0, \sum_{k=1}^{K} p_k = 1)$ に変換する関数です. 関数 $y=e^x$ は常に正なので, x_k が負でも e^{x_k} は常に正で, $\mathbf{p}$ はこれを和が 1 の確率分布になるように正規化したものになっています. $K=2$ のとき, Softmax 関数は

$$\left(\frac{e^{x_1}}{e^{x_1}+e^{x_2}}, \frac{e^{x_2}}{e^{x_1}+e^{x_2}}\right) = \left(\frac{1}{1+e^{-(x_1-x_2)}}, 1-\frac{1}{1+e^{-(x_1-x_2)}}\right)$$

ですから, Softmax 関数は式(3.86)のシグモイド関数 $p=\sigma(x)=1/(1+e^{-x})$ の多次元化と考えることができ, その意味で単に $\sigma(\mathbf{x})$ と書かれることもあります.
ここで $e^{x+a}=e^x \cdot e^a$ なので, $\mathbf{x}$ に同じ値 a を足しても,

$$\text{Softmax}\,(\mathbf{x}+a) = \left(\frac{e^{x_1+a}}{\sum_{k=1}^{K} e^{x_k+a}}, \cdots, \frac{e^{x_K+a}}{\sum_{k=1}^{K} e^{x_k+a}}\right) \tag{3.75}$$

$$= \left(\frac{e^a \cdot e^{x_1}}{\sum_{k=1}^{K} e^a \cdot e^{x_k}}, \cdots, \frac{e^a \cdot e^{x_K}}{\sum_{k=1}^{K} e^a \cdot e^{x_k}}\right) \tag{3.76}$$

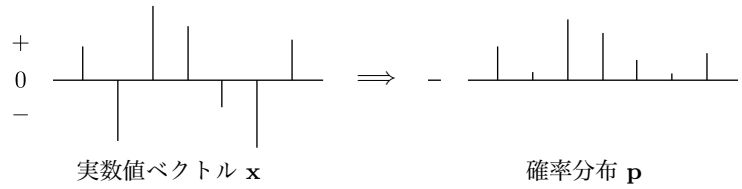


図 3.24: Softmax 関数による実数値ベクトル $\mathbf{x}$ から確率分布 $\mathbf{p}$ への変換. $\mathbf{x}$ の要素は負になることもありますが, 変換した $\mathbf{p}$ は常に正で和が 1 の確率分布になっています.

$$= \left(\frac{e^{x_1}}{\sum_{k=1}^{K} e^{x_k}}, \cdots, \frac{e^{x_K}}{\sum_{k=1}^{K} e^{x_k}} \right) = \mathbf{p} \tag{3.77}$$

となり, 得られる確率分布 $\mathbf{p}$ は変わらないことに注意してください. いっぽう, $\mathbf{x}$ に $\beta \geq 0$ をかけると, $e^{\beta x} = (e^x)^\beta$ ですから

$$\mathrm{Softmax}\,(\beta \mathbf{x}) = \left(\frac{(e^{x_1})^\beta}{\sum_{k=1}^{K} (e^{x_k})^\beta}, \cdots, \frac{(e^{x_K})^\beta}{\sum_{k=1}^{K} (e^{x_k})^\beta} \right) \tag{3.78}$$

となり, β によって異なる確率分布が得られます. $\beta = 0$ のとき $(e^x)^\beta = (e^x)^0 = 1$ ですから, 式(3.78)は $\mathbf{x}$ の値にかかわらず一様分布 $(1/K, \cdots, 1/K)$ になり, 一方 $\beta > 1$ のときは $e^{2x} = (e^x)^2, e^{3x} = (e^x)^3, \cdots$ となって差がどんどん強調され, 極端な分布に近づきます. たとえば $\mathbf{x} = (-1, 2, 3)$ のとき, $\beta = 1$ では $\mathbf{p} = (0.013, 0.265, 0.721)$, $\beta = 2$ では $\mathbf{p} = (0.0003, 0.1192, 0.8805)$, $\beta = 10$ では $\mathbf{p} = (0, 0.00005, 0.99995)$ となります. よって $\beta \to \infty$ のとき, 最も大きい x_k だけが残って

$$\lim_{\beta \to \infty} \mathrm{Softmax}\,(\beta \mathbf{x}) = (0, \cdots, 0, \overset{k}{1}, 0, \cdots, 0)$$

となるため, これは最も大きい x_k のインデックス k を返す関数 argmax と等しくなります. β が有限の場合はそれを「ソフト」にしたものと考えることができるため, Softmax という名前が付けられています.[*66] この β は, 統計力学では絶対温度 T を用いて $\beta = 1/T$ と表される**逆温度**を意味しています. β が小さい, すなわち温度 T が高く粒子が活発に動くほど粒子の確率分布は一様に近づき, 逆に β が大きい, すなわち温度 T が低い場合は, 粒子は動かずに最も確率の高い状態に「凍りつく」ことになります.

*66 Softmax 関数の名前は, 1989 年の Bridle の論文[77]で導入されたものですが, 最大値自体ではなくそのインデックスを返すので, 本当は “Softargmax” 関数と呼ぶのが正確な表現です.

3.5　単語ベクトルとその原理

3.4 節でみたように，n グラムモデルの弱点は意味を考慮していないことでした．たとえば，「月曜日」と「火曜日」は語彙を辞書順に並べると，w_{2153} と w_{1608} のようにまったく違う単語として扱われるため，たとえ「来週 月曜日」というバイグラムが 100 回現れたとしても，たまたま「来週 火曜日」がテキストに現れていなければ，p(火曜日 | 来週) は非常に低い確率になってしまいます．これは，2 章でも述べたように，単語の組み合わせに指数的な可能性があることが原因です．たとえば語彙が (少なく見積もって)10000 語=10^4 語だとしても，2 グラムは 2 単語の組み合わせで $(10^4)^2 = 10^8$ =1 億通りあり，3 グラムは $(10^4)^3 = 10^{12}$ =1 兆通りと，天文学的な組み合わせになってしまいます．テキストがいかに長くても，4 グラムや 5 グラムの可能性をすべて網羅するのはほぼ不可能で，こうした問題を一般に**データスパースネス**の問題といいます．*67 このため，n グラム言語モデルでは n グラムの頻度の多くがゼロになる，**ゼロ頻度問題**が深刻なのでした．

3.5.1　ニューラル n グラム言語モデル

この問題を解決するまったく新しいアプローチとして，2000 年にモントリオール大学の Bengio らが，**ニューラル n グラム言語モデル**を発表しました[78][79]．*68 以下で説明するように，この研究が現在の深層学習全体へと繋がっていくことになります．

Bengio らは，語彙に含まれる V 個の単語をそれぞれ独立に考える代わりに，それぞれの単語 w が K 次元 (たとえば $K=100$ 次元) の実数ベクトル $\vec{w}$ で表されるとしました．これを**単語ベクトル**，あるいは単語の**分散表現**といいます．*69

---

*67　推定すべきパラメータの組み合わせに指数的な可能性があるという意味で，これを**次元の呪い**ともいいます．

*68　実際にはこれ以前に，1980 年代から Elman や，深層学習の父ともいわれる Hinton といった認知科学者の研究があり，ニューラル n グラム言語モデルや深層学習は，それらの研究を受け継いだものです．

*69　“分散” (distributed) とは，単語や概念を表すのに V 次元のベクトルのどれかを 1，残りを 0 にする one-hot (あるいは局所) 表現のかわりに，K 次元の実数全体で表現するという意味で，その原点は Hinton など認知科学者による初期のニューラルネット研究にさかのぼります[80]．

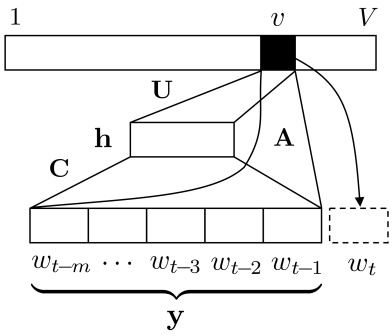


図 3.26: ニューラル n グラム言語モデルの構造. 直前の $m=(n-1)$ 単語の単語ベクトルを連結したベクトル $\mathbf{y}$ を用いて, 次の単語 $w_t=v$ の確率を計算します.

図 3.25: Wikipedia から学習された単語ベクトルの例 (一部).*70 色の濃さが値の大きさに対応しています. 2 つは微妙に違うだけで, ほとんど同じベクトルになっています.

たとえば, 「月曜日」と「火曜日」に図 3.25 のようにそれぞれ似たベクトルを割り当てることができれば, 頻度にかかわらず, p(月曜日 | 来週) と p(火曜日 | 来週) に, ほとんど同じ確率を与えることができるはずです.

具体的には, ニューラル n グラム言語モデルでは, n グラムの条件つき確率

$$p(w_t|w_{t-1}, w_{t-2}, \cdots, w_{t-m}) \qquad (m=n-1) \tag{3.79}$$

を求めるために, m 語の文脈 $w_{t-1}w_{t-2}\cdots w_{t-m}$ のそれぞれの単語ベクトルを横に連結した, $D=m\times K$ 次元のベクトル

$$\mathbf{y} = (\vec{w}_{t-1}, \vec{w}_{t-2}, \cdots, \vec{w}_{t-m}) \tag{3.80}$$

を考えます. ニューラル n グラム言語モデルでは, この $\mathbf{y}$ を使った二種類の回帰モデルを同時に用いて, 次の単語を予測します.

図 3.26 に示したように, 1 つは $\mathbf{y}$ を直接使った線形回帰モデルで, V 次元の実数ベクトル $\mathbf{x}=(x_1, \cdots, x_V)$ $(x_i \in \mathbb{R})$ を

*70 日本語 Wikipedia から学習された 100 次元の `jawiki` 単語ベクトルの, 最初の 30 次元を示しました.

$$\mathbf{x} = \mathbf{b} + \mathbf{A}\mathbf{y} \qquad (\mathbf{A}:\ V \times D \text{ 次元の行列},\ \mathbf{b}:\ V \text{ 次元のベクトル}) \tag{3.81}$$

のように計算します．もう 1 つは, $\mathbf{y}$ から射影した H 次元の隠れ層ベクトル $\mathbf{h} = \tanh(\mathbf{d} + \mathbf{C}\mathbf{y})$ を介した, 非線形な回帰モデルで

$$\mathbf{x} = \mathbf{U}\mathbf{h} = \mathbf{U}\tanh(\mathbf{d} + \mathbf{C}\mathbf{y}) \tag{3.82}$$
$$(\mathbf{U}:\ V \times H \text{ の行列},\ \mathbf{d}:\ H \text{ 次元のベクトル},\ \mathbf{C}:\ H \times D \text{ の行列})$$

と表されます．最終的に, これら 2 つの回帰モデルを足し合わせた

$$\mathbf{x} = \mathbf{b} + \mathbf{A}\mathbf{y} + \mathbf{U}\tanh(\mathbf{d} + \mathbf{C}\mathbf{y}) \tag{3.83}$$

によって次の単語 w_t を予測します．このままでは $\mathbf{x}$ は実数値ベクトルですから, 各単語の確率に直すために, 指数の肩に乗せてから総和を 1 に正規化する **Softmax 関数**

$$\mathbf{p} = \mathrm{Softmax}(\mathbf{x}) = \left(\frac{e^{x_1}}{\sum_{v=1}^{V} e^{x_v}}, \frac{e^{x_2}}{\sum_{v=1}^{V} e^{x_v}}, \cdots, \frac{e^{x_V}}{\sum_{v=1}^{V} e^{x_v}} \right) \tag{3.84}$$

を使って, w_t のとるすべての単語の可能性 $1, \cdots, V$ の確率 $\mathbf{p} = (p_1, p_2, \cdots, p_V)$ を計算します．学習テキストでの式(3.79)の n グラム確率が大きくなるように誤差逆伝搬法で学習を行い, 単語ベクトルおよび上のパラメータ $\mathbf{A}, \mathbf{b}, \mathbf{C}, \mathbf{d}, \mathbf{U}$ を最適化します．

こうして単語ベクトルを用いるニューラル n グラム言語モデルは, 学習に必要な計算量は非常に大きいものの, 表 3.5 に示したように最高性能の Kneser–Ney 言語モデルよりさらに低いパープレキシティを見せることがわかり, 自然言語処理において単語ベクトルとニューラルネットを用いることの有効性が示され,

表 3.5: 最初のニューラル n グラム言語モデル (Bengio et al. 2000) の性能 ([79]より引用). * は, 通常の n グラムとの混合モデルであることを表します．Brown は約 100 万語, Associated Press (AP) ニュースは約 1400 万語のコーパスです．

コーパス	モデル	n	PPL
Brown	ニューラル	5	**276**
	Kneser–Ney	5	321
AP News	ニューラル *	6	**109**
	Kneser–Ney	5	117

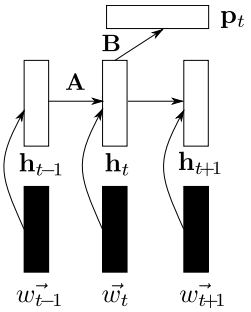


図 3.27: Mikolov が最初に用いた再帰的ニューラルネットワーク (RNN). 黒が学習される単語ベクトルです. 時刻 t での単語の確率は, 隠れ層 $\mathbf{h}_t$ を用いた Softmax 回帰によって計算されます.

2000 年代前半に言語モデルの記録を塗り替えることになりました.

3.5.2 スキップグラムと Word2Vec

その後, 上のニューラル n グラム言語モデルは改良されて, 2010 年ごろには図 3.27 のような**再帰的ニューラルネットワーク** (**RNN**) で計算されるようになりました. 単語ベクトル $\vec{w}$ を使って, RNN 言語モデルでは時刻 t での各単語の確率 $\mathbf{p}_t$ は, 実数値の行列 $\mathbf{A}$, $\mathbf{B}$ をパラメータとして次の式で計算されます.

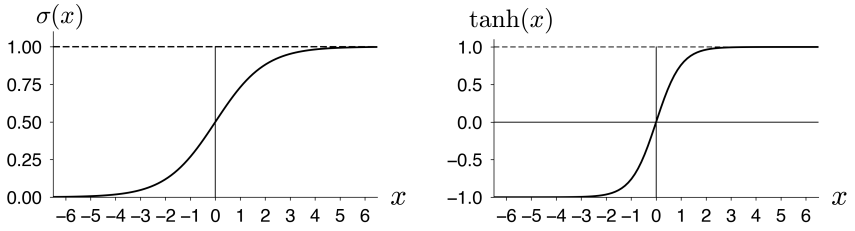
$$\begin{cases} \mathbf{h}_t = \sigma(\vec{w}_t + \mathbf{A}\mathbf{h}_{t-1}) \\ \mathbf{p}_t = \mathrm{Softmax}\,(\mathbf{B}\mathbf{h}_t) \end{cases} \tag{3.85}$$

ここで $\sigma(x)$ は実数値を $(0,1)$ の範囲の確率に変換する, 図 3.28(a) のようなシグモイド関数

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (\textbf{シグモイド関数}) \tag{3.86}$$

です. 式(3.85)のようにベクトルに適用する場合は, ベクトルの各要素に $\sigma(x)$ を適用します. さきの双曲線正接関数 $\tanh(x)$ は

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x}{e^x + e^{-x}} - \frac{e^{-x}}{e^x + e^{-x}} = \frac{1}{1+e^{-2x}} - \frac{1}{1+e^{2x}}$$



(a) $\sigma(x) = \dfrac{1}{1+e^{-x}}$ のグラフ．$\sigma(x)$ は x として実数値をとり，$(0,1)$ の範囲に収まる確率に変換します．$x=0$ のとき，確率はちょうど $1/2$ になります．

(b) $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$ のグラフ．$\tanh(x)$ は実数値をとり，$(-1,1)$ の範囲に変換します．$x=0$ で 0 になります．$\sigma(x)$ に比べ，値の増加が 2 倍になっていることに注意してください．

図 3.28: シグモイド関数 $\sigma(x)$ と双曲線正接関数 $\tanh(x)$ のグラフ．この二者には，$\tanh(x) = 2\sigma(2x)-1$ という関係があります．

$$= \sigma(2x) - \sigma(-2x) = \sigma(2x) - (1-\sigma(2x)) = 2\sigma(2x) - 1$$

と変形できますから，$\sigma(x)$ と $\tanh(x)$ の間には

$$\tanh(x) = 2\sigma(2x) - 1 \qquad (\tanh \text{ とシグモイド関数の関係}) \tag{3.87}$$

の関係があり，この 2 つの関数はスケールおよび値域を除くと，同じ関数を表しています．なお，$1-\sigma(x)$ を計算すると

$$1-\sigma(x) = 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{x}} = \sigma(-x) \tag{3.88}$$

となるため，シグモイド関数には

$$1-\sigma(x) = \sigma(-x) \qquad (\text{シグモイド関数の補関数}) \tag{3.89}$$

という関係があることがわかります．この関係は，シグモイド関数を扱う場合にこの後，頻繁に現れますので，覚えておいてください．

上のニューラル n グラム言語モデルの RNN への拡張を行ったチェコ出身の Mikolov らは 2013 年ごろ，学習された単語ベクトル $\vec{w}$ を観察して，これらが似た意味の単語について似たベクトルとなるだけでなく，ベクトルの引き算が意

味を持つようになっていることを発見しました.

たとえば, 図 3.29 左に示したように, “man” ベクトルから “woman” ベクトルに向かうベクトル $\overrightarrow{\text{woman}} - \overrightarrow{\text{man}}$ や $\overrightarrow{\text{aunt}} - \overrightarrow{\text{uncle}}$, $\overrightarrow{\text{queen}} - \overrightarrow{\text{king}}$ はほとんど同じ方向を向いており, これは「女性」を示すベクトルと考えられます. よって, “king” にこの女性ベクトルを足せば “queen” になる, すなわち

$$\overrightarrow{\text{king}} + (\overrightarrow{\text{woman}} - \overrightarrow{\text{man}}) = \overrightarrow{\text{queen}} \tag{3.90}$$

がほぼ成り立ちます.

同様に “king” → “kings”, “child” → “children” も「複数形」を表すベクトルとなっているため, “queen” にこのベクトルを足すと “queens” になる, すなわち図 3.29 右のように

$$\overrightarrow{\text{queen}} + (\overrightarrow{\text{kings}} - \overrightarrow{\text{king}}) = \overrightarrow{\text{queens}} \tag{3.91}$$

が成り立ちます. こうした関係が成り立つ理由については, この後 3.5.4 節で詳しく説明します.

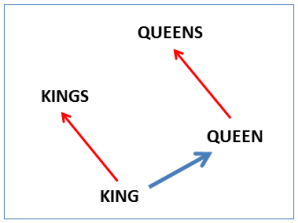


図 3.29: 単語ベクトルの間に成り立っている関係の概念図. Mikolov らの原論文[81]より引用.

こうした性質は自然言語処理一般にきわめて有用なため, 単語ベクトルのこの性質をより効率的に学習するために, Mikolov らは**連続的単語集合** (CBOW) と**スキップグラム**という, より単純化されたモデルを提案しました. この 2 つの方法はまとめて **Word2Vec** とよばれ, 以下で説明するアルゴリズムの C 言語による効率的な実装が公開されています *71

単語ベクトルの計算 Word2Vec の内部について説明する前に, まず, 実際のテキストで単語ベクトルを計算して, こうしたことを確かめてみましょう. ここでは, `gensim` にある Word2Vec の計算の標準的なパッケージを使用することにします *72. こうした単語ベクトルの計算のための 100MB の小さいコーパスであ

*71 `https://code.google.com/archive/p/word2vec/`. 原論文[63]では手法はスキップグラムと呼ばれており, `word2vec` はその実装を指していますが, わかりやすさのため, 本書では大文字で Word2Vec と表記しています.

*72 パッケージを使用しない単語ベクトルの計算法については, この後の 3.5.4 節で説明します.

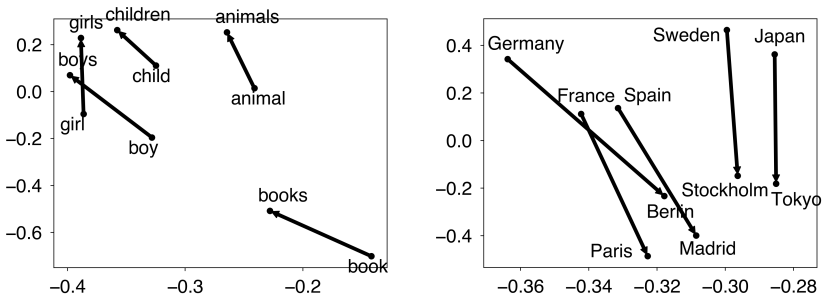


図 3.30: 単語ベクトルの差の間に成り立っている関係. 左では「複数形」の関係が, 右では「首都」の関係が, ほぼ同じ方向のベクトルとして表されています.

る日本語 text8 コーパス (83 ページ) `ja.text8` を使うと, 100 次元の単語ベクトルは次のようにして計算できます. あらかじめ, `% sudo pip install gensim` などとして `gensim` パッケージをインストールしておいてください. この学習には, 執筆時の環境では 2 分程度かかります.

```
from gensim.models import word2vec
text = word2vec.Text8Corpus("data/ja.text8")
vectors = word2vec.Word2Vec (text, size=100, \
                             min_count=10, window=10)
vectors.wv.save_word2vec_format ("model/ja.text8.vec", \
                                 binary=False)
```

この上で, 次のようなコードを書くと, 単語ベクトルが似ている言葉を出力することができます.

```
import numpy as np
def similars (vectors, source, N=10):
    target = vectors[source]
    scores = []; words = []; shown = 0
    for word,vector in vectors.items():
        scores.append (cosine(target, vector))
        words.append (word)
    for word,score in sorted (zip(words,scores), \
                      key=lambda x: x[1], reverse=True):
        if word != source:
            print ('%s -> %.4f' % (word, score))
            shown += 1
```

```
            if shown > N:
                break

    def cosine (x,y):
        return np.dot(x,y) / (norm(x) * norm(y))

    def norm (x):
        return np.sqrt (np.dot (x,x))
```

いくつかの単語について, 単語ベクトルが似ている語をコサイン類似度とともに表示してみましょう.

```
  similars(vectors, "太陽")
⇒ 恒星 -> 0.8004
  太陽系 -> 0.7675
  シリウス -> 0.7515
  土星 -> 0.7493
  銀河 -> 0.7336
  星 -> 0.7312
  地球 -> 0.7298
  火星 -> 0.7219
  超新星 -> 0.7151
  星雲 -> 0.7124
  海王星 -> 0.7095
```

```
  similars(vectors, "少年")
⇒ 少女 -> 0.8339
  高校生 -> 0.7346
  小学生 -> 0.7230
  中学生 -> 0.7030
  青年 -> 0.6916
  同級生 -> 0.6266
  若い -> 0.5787
  主人公 -> 0.5785
  天才 -> 0.5767
  幼児 -> 0.5745
  憧れ -> 0.5744
```

たった 100MB のテキストから学習したにもかかわらず, 単語の類似度を非常によく反映していることがわかります. 下の例の左側のように, 類似語があまり正しくない場合もありますが, これは ja.text8 のテキストが 100MB と, かなり小さいためです. 筆者が, ja.text8 と同じ方法で準備したその 10 倍の 1GB のテキストを, サポートサイトに ja.text9 として置いておきました [*73]. これを使って同様に単語ベクトルを学習すると, 類似語は右のように, かなり直感的になります. ただし, この学習には 40 分程度かかりますので注意してください.

```
text9 = word2vec.Text8Corpus("data/ja.text9")
vectors9 = word2vec.Word2Vec (text9, size=400, \
                              min_count=10, window=10)
vectors9.wv.save_word2vec_format ("model/ja.text9.vec", \
                                  binary=False)
```

*73 執筆時で Wikipedia 日本語版の記事は合計で 3.4GB 程度ありますので, これは日本語 Wikipedia 全体の約 1/3 に相当しています.

```
  ja.text8 から学習した場合
  similars (vectors, "アパレル")
⇒ プランニング -> 0.7297
  アサヒ -> 0.6975
  最大手 -> 0.6842
  量販 -> 0.6604
  アミューズメント -> 0.6579
  老舗 -> 0.6573
  コンサルティング -> 0.6560
  服飾 -> 0.6500
  プロダクト -> 0.6486
  フード -> 0.6461
  デジキューブ -> 0.6414
```

```
  ja.text9 から学習した場合
  similars (vectors9, "アパレル")
⇒ ファッション -> 0.6495
  ジュエリー -> 0.6451
  ブティック -> 0.6376
  ブランド -> 0.6114
  雑貨 -> 0.6112
  メンズ -> 0.6082
  衣料 -> 0.6049
  アパレルメーカー -> 0.5955
  アクセサリー -> 0.5940
  プレタポルテ -> 0.5837
  文具 -> 0.5781
```

なお, ベクトルの次元は通常は 100〜1000 次元程度で, ja.text8 のように小さいデータなら 100 次元程度, 大きなデータなら 500〜700 次元程度を指定するのが普通です [*74].

単語の比例関係　図 3.30 に示したような単語ベクトルの「引き算」は, king : queen = boy : □ となる□を求めるのに, 図 3.29 のように $\overrightarrow{\mathrm{boy}}+(\overrightarrow{\mathrm{queen}}-\overrightarrow{\mathrm{king}})$ を計算すればよい, ということですから, □に当てはまる語は, 次のようなコードで計算することができます.

```
def contrast (vectors, a, b, x, N=10):
    scores = []; words = []; shown = 0
    target = vectors[x] + (vectors[b] - vectors[a])
    for word,vector in vectors.items():
        scores.append (cosine(vector, target))
        words.append (word)
    for word,score in sorted (zip(words,scores), \
                        key=lambda x: x[1], reverse=True):
        if (word != x) and (word != b):
            print ('%s -> %.4f' % (word, score))
            shown += 1
        if shown > N:
            break
```

*74　研究レベルでは「右打ち切り可能」, すなわち左から重要な次元が並んでおり, どこで打ち切ってもそこまでで最適な性能が出るように単語ベクトルを学習する方法も提案されています[82]. また, 3.5.4 節で説明する行列分解による単語ベクトルは, 主成分分析を利用しているため, 同様に最も重要な固有ベクトルから順に用いた単語ベクトルを計算することができます.

いくつかの例について単語の「比例関係」を計算してみると, 次のようになります.

```
    contrast (vectors,
    "日本", "東京", "フランス")
⇒ パリ -> 0.6875
  ウィーン -> 0.5536
  ベルリン -> 0.5512
  ロンドン -> 0.5363
  ニュルンベルク -> 0.5306
  ミュンヘン -> 0.5276
  トゥールーズ -> 0.5151
  オーストリア -> 0.5114
  ハンブルク -> 0.5023
  プラハ -> 0.4984
  近郊 -> 0.4964
```

```
    contrast (vectors,
    "王様", "女王", "男子")
⇒ 女子 -> 0.6027
  アテネ -> 0.5622
  大公 -> 0.5466
  爵位 -> 0.5447
  金メダル -> 0.5366
  ダブルス -> 0.5286
  オリンピック -> 0.5241
  即位 -> 0.5201
  称号 -> 0.5180
  王妃 -> 0.5170
  王位 -> 0.5095
```

こちらも, 学習するテキストを増やすとより正確になります [*75]. こうした単語ベクトルの演算はすべて, 単語ベクトルの長さを 1 に規格化した上で行っている (単位超球面上のベクトルとして計算を行っている) ことに注意してください.

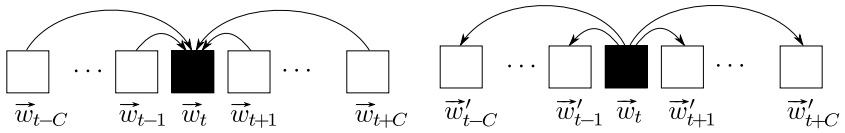
```
    contrast (vectors9,
    "日本", "東京", "フランス")
⇒ パリ -> 0.5812
  リヨン -> 0.5394
  マルセイユ -> 0.5321
  トゥールーズ -> 0.5123
  ニース -> 0.5098
  ストラスブール -> 0.4985
  ナント -> 0.4955
  ディジョン -> 0.4722
  ブリュッセル -> 0.4655
  ボルドー -> 0.4468
  マドリード -> 0.4367
```

```
    contrast (vectors9,
    "日本", "聖子", "アメリカ")
⇒ リンダ -> 0.4708
  ジャネット -> 0.4679
  マライア -> 0.4667
  マリリン -> 0.4603
  ジョニー -> 0.4508
  ビリー -> 0.4478
  テイラー -> 0.4402
  ロジャー -> 0.4344
  パウエル -> 0.4299
  タウンゼント -> 0.4274
  ディーン -> 0.4265
```

このように, 単語ベクトルは大変興味深い性質を持っていますが, このとき問題になるのは,

- 学習された単語ベクトルには, なぜこうした性質が成り立つのか

*75 松田聖子に対応するアメリカの歌手として一位になった “リンダ” は, 形態素解析の都合で切れていますが, 「リンダ・ロンシュタット」のことではないかと考えられます.



(a) 連続的単語集合 (CBOW) の場合　　(b) スキップグラム (skip-gram) の場合

図 3.31: `word2vec` での単語ベクトルの学習方法. 単語ベクトル $\vec{w}_t$ は, (a) 周囲の単語ベクトルから自分を予測できるか, または (b) 自分から周囲の単語ベクトルを予測できるか, のどちらかを目的にして学習されます.

- 単語ベクトルはどうやって学習されるのか

という点でしょう. この 2 つの点は密接に関連していますので, まず, 単語ベクトルがどうやって学習されるかについてみていくことにしましょう.

3.5.3　単語ベクトルの学習

(a) 連続的単語集合 (CBOW)

図 3.26 のニューラル n グラム言語モデルでは実験によって, 式(3.82)の隠れ層 $\mathbf{h}$ を使わず, 式(3.81)の $\mathbf{x}$ からの対数線形モデルだけでも十分な性能が出ることがわかっていました. そこで, $\mathbf{x}$ として文脈語の単語ベクトルを連結する代わりに, 図 3.31(a) に示したように共通する同じ次元のベクトルに足し合わせ, この $\mathbf{x}$ による式(3.81)の対数線形モデルで次の単語 w_t を予測すれば, より簡単に単語ベクトルを学習することができます. 文脈 $w_{t-m}, \cdots, w_{t-1}$ の順番を考えず, それらを $\mathbf{x}$ に足し込むため, これは単語の順番を考慮せず, 袋 (bag) づめにした集合とみる単語集合 (5 章) の連続版とみなすことができるため, これを**連続的単語集合** (continuous bag of words, **CBOW**) モデルといいます.

上では文脈として, 直前の $m=n-1$ 単語を用いましたが, 現実には単語の予測には, その後に続く文脈も使った方がよりよく w_t を予測できると考えられます. たとえば,

先月 自民党 は □

に続く□の単語には「国会」「ホームページ」「永田町」といった, いくつもの可能性がありますが, その後に続く単語が

先月 自民党 は □ に 法案 を

とわかっていれば, □の単語はほぼ「国会」と決定することができるでしょう. よって実際には, CBOW は図 3.31(a) のように, 後続する単語 $w_{t+1}, w_{t+2}, \cdots, w_{t+C}$ も使って計算します. すなわち式で書けば,

$$\begin{cases} \mathbf{x} = \sum_{i \in n(t)} \vec{w}_i \qquad (n(t) = \{t-C, \cdots, t-1, t+1, \cdots, t+C\}) \\ \mathbf{p}_t = \mathrm{Softmax}\,(\mathbf{A}\mathbf{y}) \end{cases} \tag{3.92}$$

という形になります. *76

CBOW では文脈の単語の順番を考慮しないため, n グラムモデルと異なり組み合わせ爆発 (次元の呪い) が発生しないので, 文脈の長さ C を大きくとることができます. 通常は C は 10 以下の値を用いますが, C を大きく取りすぎると「意味がぼやける」ため, 適切な値に設定する必要があります. また, 文脈として前後の単語ではなく, 係り受け関係にある (離れている可能性もある) 語を使うこともできます. その場合には単語ベクトルとして, より文法的な機能が重視されたベクトルが得られることになり, 構文解析などに有用であることが確かめられています[83]. このように, 使う「文脈」は, ベクトルの用途によって本来異なることに注意してください. 通常のスキップグラムのように前後の単語を使う場合でも, Cが小さいほど文法的な関係が, Cが大きいほど意味的な関係が重視された単語ベクトルが学習されることになります.

(b) スキップグラム

上の CBOW では周囲の単語ベクトルの平均から, 中心の単語 w_t を予測することを目的として単語ベクトルを学習していました. もう一つのアプローチとして, 図 3.31(b) のように, w_t から逆に周囲の単語を予測する, という方法も考えられます. つまり,

*76 なお, このことで n グラムと異なり, CBOW は単語列の生成モデルではなくなってしまいます. 数学的には, 周囲が与えられた場合の中心の語の確率が与えられている, イジングモデル[26, 31 章]と同様なマルコフ確率場を考えていることになります. 生成モデルからマルコフ確率場への定式化の変化は, 深層学習による GPT-3 のような現在の強力なマスク化言語モデルにも受け継がれています.

$$\prod_{t=1}^{T} \prod_{j \in n(t)} p(w_j|w_t) \tag{3.93}$$

を最大化することが目的となります．このとき，

$$\begin{cases} p(w_j|w_t) = \dfrac{\exp(\vec{w}'_j \cdot \vec{w}_t)}{Z} \\ Z = \sum_{v=1}^{V} \exp(\vec{v}'_j \cdot \vec{w}_t) \end{cases} \tag{3.94}$$

です．式(3.94)では，w_t と周辺の語とのバイグラムを，必ずしも隣りあっていない，何語かスキップした場合も含めて考えるため，これを**スキップグラム** (skip-gram) といいます．なお，スキップグラムでは目的とする単語ベクトル $\vec{w}_t$ と，周辺語のベクトル $\vec{w}'_j$ は別のものを用います．すなわち，各単語 w について $\vec{w}$ と $\vec{w}'$ の二種類のベクトルを考えることになります．

スキップグラムでは，CBOW と異なり，周辺語の単語ベクトルを $\mathbf{x}$ に平均化することなく，$\vec{w}_t$ と $\vec{w}'_j$ を直接比較して単語ベクトルを最適化するため，より単語ベクトルの表現力が高まることが期待されます．

ただし，ナイーブに式(3.94)を用いると，計算量が非常に大きくなってしまいます．これは，正規化定数である Z[*77] が，$\vec{w}_t$, $\vec{w}'_j$ が最適化で変わるごとにすべて計算し直さねばならず，この和は語彙に含まれる V 個の単語すべてにわたる和で，通常 V は少なくても 1 万，多いと 10 万からそれ以上にもなるからです．

この問題に対する解決法として，階層的 Softmax を使う方法と負例サンプリングを使う方法の 2 つがありますが，現在主に使われているのは後者ですので，以下でみていくことにしましょう．

負例サンプリング　式(3.94)のスキップグラムで単語ベクトルを効率的に学習する方法として，機械学習で用いられている Noise Contrastive Estimation [84] の考え方を用いて，

- 実際に出現した語 c の確率を大きく，かつ

*77　統計力学ではこうした和は分配関数とよばれ，慣習的に Z が用いられますので，ここでもそれを踏襲しています．この場合，exp() の中の $\vec{w}'_j \cdot \vec{w}_t$ が単語ベクトルどうしの内積がもつエネルギー，すなわち「ハミルトニアン」ということになります．

- それ以外の任意の語 c' の確率を小さく

することが考えられます. すなわち,

$$L = \prod_{c \in n(w)} \sigma(\vec{c} \cdot \vec{w}) \times \prod_{c' \notin n(w)} (1 - \sigma(\vec{c}' \cdot \vec{w})) \tag{3.95}$$

を最大化することを考えます. ただし, 式(3.95)の第 2 項は w の周囲に出現したごく少数の語以外の, ほとんどすべての単語に関する積ですので, これを期待値で置き換えて, $1-\sigma(x) = \sigma(-x)$ ですから

$$L = \prod_{c \in n(w)} \sigma(\vec{c} \cdot \vec{w}) \times \mathbb{E}_{c \sim p(c)} \Big[\sigma(-\vec{c} \cdot \vec{w}) \Big] \tag{3.96}$$

としてみましょう. さらに後半の期待値を, 単語分布 $p^*(c)$ からの k 個のサンプルを用いたモンテカルロ積分 (66 ページ脚注) で置き換えて,

$$= \prod_{c \in n(w)} \sigma(\vec{v} \cdot \vec{w}) \times \frac{1}{k} \sum_{i=1}^{k} \sigma(-\vec{c}^{(i)} \cdot \vec{w}) \quad ; \quad c^{(i)} \sim p^*(c) \tag{3.97}$$

を最大化することにします. 1 回の最適化では k 個の $c^{(1)}, \cdots, c^{(k)}$ しか負例として考慮しませんが, 一般にこの最適化を繰り返すため, 学習の過程ではさまざまな単語 c が負例として登場することに注意してください. 負例, すなわちモンテカルロサンプルの数 k は, 小さいコーパスの場合は 5 ～ 20 個程度, 大きなコーパスでは 2 ～ 5 個程度でよいことが原論文で報告されています.

負例をサンプリングする分布 $p^*(c)$ としては, 一様分布 $1/V$ やユニグラム分布 $p(c)$ など, さまざまな可能性があります. ただし, 一様分布では「奏覧」や「卯原内」といった極端に稀な単語も同等に出現してしまいますし [*78], いっぽうユニグラム分布を使うと, 3.2 節でみたように「が」「の」といった, ごく一部の機能語ばかりが確率が高いためにサンプリングされることになってしまいます. 実験によれば, この間をとって, ユニグラム分布を 3/4 乗して平滑化した

*78 3.2 節の Zipf の法則によって, こうした頻度の低い語は非常に多く存在することに注意してください. 実際には, 語彙のほとんどはこうした語からなっているため, 一様分布からのサンプルはほとんどが稀な単語で占められてしまうことになります.

$$p^*(v) = \frac{p(v)^{3/4}}{Z} \qquad \left(Z = \sum_{v=1}^{V} p(v)^{3/4}\right) \tag{3.98}$$

を使うとよい結果になることが確かめられています.

さらに, そもそも「の」「が」といった意味の薄い頻出語との共起を抑えるために, Word2Vec では

$$r(v) = \max\left(1 - \sqrt{\frac{t}{p(v)}}, 0\right) \tag{3.99}$$

の確率で, 共起の計算前に文から単語を削除するヒューリスティックが使われています.*79 ここで t は 10^{-5} 程度の閾値で, 単語の確率 $p(v)$ がこれより低い単語は削除されず, これより高いと一定の割合で削除されることになります. この関数の形については, 4 章の図 4.3 を参照してください. たとえば, Brown コーパスでは $p(\texttt{the})=0.069$, $p(\texttt{hexagonal})=0.00000198$ なので, $t=10^{-5}$ のとき

$$\begin{cases} r(\texttt{the}) = \max\left(1 - \sqrt{\dfrac{10^{-5}}{0.069}},\ 0\right) = 0.962 \\ r(\texttt{hexagonal}) = \max\left(1 - \sqrt{\dfrac{10^{-5}}{0.00000198}},\ 0\right) = \max\left(-1.247, 0\right) = 0 \end{cases}$$

となり, 式(3.99)のルールでは “`the`” は 96%の確率で文から削除されることになり, 一方 “`hexagonal`” はまったく削除されないことになります.

Word2Vec では, テキストの単語をこうしてあらかじめ確率的に削除してから計算を行っています. これにより, “`the`” のような単語が削除されて共起窓が実質的に広がるため, より意味を考慮した共起データを得ることができ, 性能が改善することが確かめられています[85].

*79　原論文[63]では $p(v)$ は v の頻度となっており, その場合は式(3.99)は無意味な値となるため, 誤っていることに注意が必要です. 実際の `word2vec.c` や gensim の実装ではまた違った式が使われており, これに理論的な根拠があるわけではありません. より理論的な方法については, **??**節の SIF 単語重みの議論を参照してください.

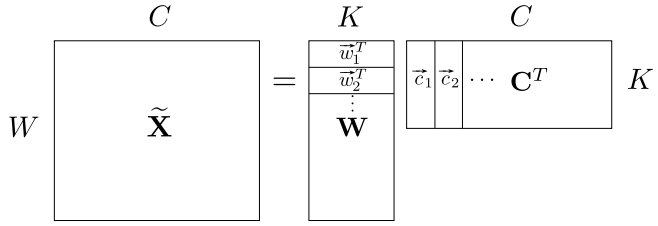


図 3.32: 行列の特異値分解 (SVD) によるニューラル単語ベクトルの学習. Word2Vec で学習される単語ベクトルは, Shifted PPMI (本文参照) を要素とする行列 $\widetilde{\mathbf{X}}$ を $\widetilde{\mathbf{X}} \simeq \mathbf{W}\mathbf{C}^T$ と特異値分解して得られるベクトルと, 数学的に等価です.

3.5.4 Word2Vec と行列分解

前の節で計算したニューラル単語ベクトルは, 数学的には何を計算していることになるのでしょうか. Levy と Goldberg [86]は, word2vec で得られる単語ベクトルは, 以下に示すように, データから計算されるある行列の主成分分析 (特異値分解) と数学的に等価であることを示しました. これにより, 実はニューラル単語ベクトルは, 行列の特異値分解で学習することができます. ここからは行列の計算を頻繁に用いますので, 線形代数に慣れていない方は, 本章の文献案内を参照してください.

Word2Vec のスキップグラムで最大化している目的関数は, 式(3.95) でした. この式を最適化することで, 単語ベクトル $\vec{v}$ と周辺語ベクトル $\vec{c}$ が計算できますが, 式(3.95)で, $\vec{v}$ と $\vec{c}$ はつねに $\vec{v} \cdot \vec{c} = \vec{v}^T \vec{c}$ の形でしか現れません. すべての $\vec{v}^T \vec{c}$ の組み合わせは, 図 3.32 に示したように $\vec{w_1}^T, \vec{w_2}^T, \cdots$ を縦に並べた行列を $\mathbf{W}$, また $\vec{c_1}, \vec{c_2}, \cdots$ を横に並べた行列を $\mathbf{C}^T$ とおいて

$$\mathbf{X} = \mathbf{W}\mathbf{C}^T \tag{3.100}$$

の要素になっています. よって, 式(3.95)は本質的にこの行列 $\mathbf{X}$ を最適化しており, それを式(3.100)のように行列分解すれば, 単語ベクトルの行列 $\mathbf{W}$ と周辺語ベクトルの行列 $\mathbf{C}$ が得られる, ということがわかります. それでは, この行列 $\mathbf{X}$ はどんな行列でしょうか.

$\mathbf{X}$ の (v, c) 要素は内積 $\vec{v} \cdot \vec{c}$ ですから, これを $X(v, c) = x = \vec{v} \cdot \vec{c}$ とおく

と, ある x についての式(3.95)の目的関数 $L(x)$ は,

$$\log L(x) = n(w,c)\log\sigma(x) + k\cdot n(w)\,p(c)\log\sigma(-x) \tag{3.101}$$

となります. ここでシグモイド関数 $\sigma(x)$ の微分について, 一般に

$$\begin{cases} \sigma(x)' &= \left(\dfrac{1}{1+e^{-x}}\right)' = -\dfrac{-e^{-x}}{(1+e^{-x})^2} = \sigma(x)\sigma(-x) \\ \sigma(-x)' &= \left(\dfrac{1}{1+e^{x}}\right)' = -\dfrac{e^{x}}{(1+e^{x})^2} = -\sigma(x)\sigma(-x) \end{cases} \tag{3.102}$$

が成り立つことに注意しましょう. よって,

$$\begin{cases} \log\sigma(x)' &= \dfrac{1}{\sigma(x)}\cdot\sigma(x)\sigma(-x) = \sigma(-x) \\ \log\sigma(-x)' &= \dfrac{1}{\sigma(-x)}\cdot -\sigma(x)\sigma(-x) = -\sigma(x) \end{cases} \tag{3.103}$$

です. これを使えば, 式(3.101)が最大になる点で x に関する勾配は 0 になりますから, x で微分して 0 とおけば

$$\frac{\partial}{\partial x}\log L(x) = n(w,c)\,\sigma(-x) + k\cdot n(w)\,p(c)\cdot(-\sigma(x)) = 0 \tag{3.104}$$

が成り立ちます. よって, 式(3.89)のように $\sigma(-x)=1-\sigma(x)$ でしたから,

$$\begin{aligned} &n(w,c)(1-\sigma(x)) - k\cdot n(w)\,p(c)\sigma(x) = 0 \\ &\therefore\quad \sigma(x) = \frac{n(w,c)}{n(w,c)+k\cdot n(w)\,p(c)} \quad (\equiv y) \end{aligned} \tag{3.105}$$

となります. 式(3.105)の 2 行目の右辺を y とおくと,

$$\sigma(x) = \frac{1}{1+e^{-x}} = y \quad \text{を解いて} \quad x = -\log\left(\frac{1}{y}-1\right) \tag{3.106}$$

を式(3.105)の 2 行目に代入して整理すれば,

$$x = -\log\left(\frac{n(w,c)+k\cdot n(w)\,p(c)}{n(w,c)}-1\right) \tag{3.107}$$

$$= \log \frac{n(w,c)}{k \cdot n(w)\, p(c)} = \log \frac{\frac{n(w,c)}{N}}{\frac{n(w)}{N}\, p(c)} - \log k$$
$$= \log \frac{p(w,c)}{p(w)\, p(c)} - \log k$$

となることがわかります．すなわち，x は w と c に対する，式(3.16)でも使った**自己相互情報量** (**PMI**)

$$\mathrm{PMI}\,(w,c) = \log \frac{p(w,c)}{p(w)\, p(c)} \tag{3.108}$$

を，$\log k$ だけシフトしたものになっています．これから，Word2Vec で学習している式(3.100)の行列 $\mathbf{X}$ は，

$$\begin{aligned} X(w,c) &= \mathrm{PMI}\,(w,c) - \log k \\ &= \log \frac{p(w,c)}{p(w)\, p(c)} - \log k \end{aligned} \tag{3.109}$$

を要素とする行列だということがわかりました．特に，負例数 $k=1$ のときは $\log k = \log 1 = 0$ ですから，$\mathbf{X}$ は単に $\mathrm{PMI}\,(w,c)$ を並べた行列となります．

ただし，式(3.109)の行列は，ほとんどを占める $p(w,c)=0$ の場合に値が $\log p(w,c) = -\infty$ になってしまいます．要素がすべて埋まると計算量も増えてしまうため，実際に現れる (w,c) のペアのほとんどは $\mathrm{PMI}\,(w,c) > 0$ である（だからこそ出現した）ことから，式(3.109)で値が非負の場合だけを考えて [*80]，

$$\widetilde{X}(w,c) = \max\left(\log \frac{p(w,c)}{p(w)\, p(c)} - \log k, 0\right) \tag{3.110}$$

を用いることにします．これを，SPPMI (Shifted Positive PMI) とよびます [*81]．

---

*80 PMI が負の場合を考えないことで，正確には目的関数は式(3.101)とは異なってしまいますが，実験的には以下で示すように，これは Word2Vec のかなり良い近似になっていることがわかっています．

*81 PPMI はこの研究で初めて現れたわけではなく，5 章で扱う潜在意味解析 (LSA) の文脈で，2007 年に Bullinaria らによって提案され[87]，他のさまざまな単語ベクトルに比べて，PPMI による単語ベクトルとコサイン距離による比較が最も高い性能を持つことが報告されていました．2012 年には SVD による次元圧縮も試みられており[88]，Word2Vec と本質的に同じベクトルが，深層学習より以前にすでに提案されていたといえます．

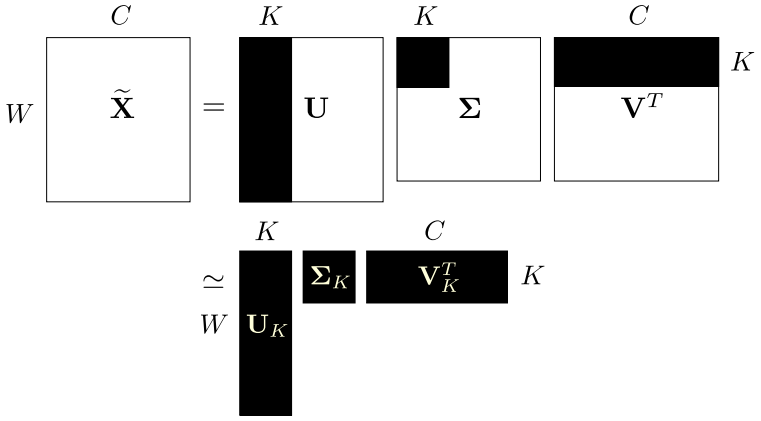


図 3.33: SPPMI 行列 $\widetilde{\mathbf{X}}$ の特異値分解 (SVD) による次元削減の様子.

SPPMI を並べた行列 $\widetilde{\mathbf{X}}$ は, カウントが 0 のエントリ (w, c) および式(3.109)が負になるエントリはすべて値が 0 になるため, 非常に疎になることに注意してください. 実際, `ja.text8.txt` で窓幅 10 の場合, 非 0 のエントリは 1% (20872503/2027160576) にすぎず, 99%のエントリが 0 になりました. $\widetilde{\mathbf{X}}$ から式(3.100)の $\mathbf{W}, \mathbf{C}$ を求めるには, 疎行列に対する**特異値分解** (singular value decomposition, SVD) を使うと高速に計算することができます. SVD は行列の対角化を正方行列でない場合に拡張したもので, 行列 $\widetilde{\mathbf{X}}$ を図 3.33 の上段のように

$$\widetilde{\mathbf{X}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{3.111}$$

と 3 つの行列 $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^T$ の積に分解します. ここで $\mathbf{U}, \mathbf{V}$ はそれぞれ $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T$ および $\widetilde{\mathbf{X}}^T\widetilde{\mathbf{X}}$ の固有ベクトル, $\mathbf{\Sigma}$ は対応する固有値の平方根 (特異値) $\sigma_1, \sigma_2, \cdots, \sigma_r$ (r は $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T, \widetilde{\mathbf{X}}^T\widetilde{\mathbf{X}}$ のランク) を並べた対角行列

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & & & \mathbf{O} \\ & \sigma_2 & & \\ & & \ddots & \\ \mathbf{O} & & & \sigma_r \end{pmatrix} \tag{3.112}$$

です．このうち特異値の大きい方から上位 K 個をとって，図 3.33 の下段のように

$$\widetilde{\mathbf{X}} \simeq \mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T \tag{3.113}$$

とする近似は，$\widetilde{\mathbf{X}}$ との二乗誤差を最小にする近似になっています[]．式(3.113)は，$\mathbf{\Sigma}$ は対称行列なので $\mathbf{\Sigma}^T = \mathbf{\Sigma}$ で，一般に $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ ですから，

$$\begin{aligned} \widetilde{\mathbf{X}} &\simeq \mathbf{U}_K \mathbf{\Sigma}_K^{\frac{1}{2}} \mathbf{\Sigma}_K^{\frac{1}{2}} \mathbf{V}_K^T \\ &= \left(\mathbf{U}_K \mathbf{\Sigma}_K^{\frac{1}{2}}\right)\left(\mathbf{V}_K \mathbf{\Sigma}_K^{\frac{1}{2}}\right)^T \quad (= \mathbf{W}\mathbf{C}^T) \end{aligned} \tag{3.114}$$

とも書くことができます．よって，式(3.100)と見比べて

$$\mathbf{W} = \mathbf{U}_K \mathbf{\Sigma}_K^{\frac{1}{2}}, \quad \mathbf{C} = \mathbf{V}_K \mathbf{\Sigma}_K^{\frac{1}{2}} \tag{3.115}$$

とおけば，$\mathbf{W}, \mathbf{C}$ を得ることができます [*82]．ここで $\mathbf{\Sigma}_K^{\frac{1}{2}}$ は，$\mathbf{\Sigma}$ の対角成分の平方根をとった行列

$$\mathbf{\Sigma}^{\frac{1}{2}} = \begin{pmatrix} \sqrt{\sigma_1} & & & \mathbf{O} \\ & \sqrt{\sigma_2} & & \\ & & \ddots & \\ \mathbf{O} & & & \sqrt{\sigma_r} \end{pmatrix} \tag{3.116}$$

です．Python では，

```
from scipy.sparse.linalg import svds
from pylab import *
U,S,V = svds(X)
W = np.dot (U, diag(sqrt(S)))
C = np.dot (V, diag(sqrt(S)))
```

のように実行すれば，$\mathbf{W}, \mathbf{C}$ を計算することができます．

単語ベクトルの計算　式(3.108)の PMI は，式(3.107)から，頻度 $n(w,c)$, $n(w)$, $n(c)$ を用いて

---

*82　行列 $\widetilde{\mathbf{X}}$ の各行，すなわち各単語の共起情報を格納したベクトルの間の内積は $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T$ で計算することができます．$\widetilde{\mathbf{X}} \simeq \mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T$ ですから，これは $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T = \mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T (\mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T)^T = \mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T \mathbf{V}_K \mathbf{\Sigma}_K^T \mathbf{U}_K^T = (\mathbf{U}_K \mathbf{\Sigma}_K)(\mathbf{V}_K \mathbf{\Sigma}_K)^T$ となり，$\mathbf{W} = \mathbf{U}_K \mathbf{\Sigma}_K, \mathbf{C} = \mathbf{V}_K \mathbf{\Sigma}_K$ とするのが $\widetilde{\mathbf{X}}$ の各行間の内積を保存する意味では理論的に最適です．しかし，式(3.114)のように直接行列分解を近似する方が，さまざまな意味的タスクにおいて精度が高いことが確かめられています[85]．

$$\text{PMI}\,(w,c) = \log\frac{p(w,c)}{p(w)\,p(c)} = \log\frac{p(w,c)/N}{n(w)/N\cdot n(c)/N} \tag{3.117}$$
$$= \log n(w,c) - \log n(w) - \log n(c) + \log N$$

で計算することができます．式(3.110)から，この値が $\log k$ より大きくないと値が 0 になり，文脈語 c が観測されなかったことになってしまいますので，[85] では実験の結果，$k=1$ すなわち $\log k=0$ で PMI をそのまま使う方が意味的タスクにおいて高性能となることが確かめられています．また式(3.108)は，条件つき確率の定義から

$$\text{PMI}\,(w,c) = \log\frac{p(w,c)}{p(w)\,p(c)} = \log\frac{p(c|w)}{p(c)} \tag{3.118}$$

とも書くことができることに注意してください．すなわち $\text{PMI}\,(w,c)$ は，「w の周辺に c が現れる確率」と「c が一般的に出現する確率」の比の対数になっています [*83]．このとき，c が稀な単語で $p(c)$ が非常に小さいと，PMI の値が非常に大きくなってしまいます．これを避けるため，$p(c)$ としては負例サンプリングでも用いた，式(3.98)で平滑化した $p^*(c)$ を用いるとよいでしょう[85]．あらかじめ式(3.98)の対数 $\log p^*(c)$ を計算しておけば，平滑化された式(3.118)は

$$\log\frac{p(c|w)}{p^*(c)} = \log\frac{n(w,c)/n(w)}{p^*(c)} = \log n(w,c) - \log n(w) - \log p^*(c) \tag{3.119}$$

で求めることができます．

こうして計算した単語ベクトルの例を，図**??**に示しました．この計算は，サポートページの `pmivec.py` を使って，

```
% pmivec.py K text model
```

のようにして実行することができます．なお，共起行列 $\widetilde{\mathbf{X}}$ が巨大になる場合は，ランダム射影を用いて SVD を効率的に行うことのできる redsvd []のような方法を使えば，データが大きくても高速に単語ベクトルを計算することができます．

自己相互情報量と言語モデル ▽　SGNS は $\text{PMI}\,(w,c)$ を近似していることがわかりましたが，言語の生成モデルとしては，これは何を意味しているのでしょう

*83　これは，**対数尤度比**ともよばれています．

確率	$k=$飼育	$k=$彼女	$k=$動物	$k=$本陣
$p(k \mid 犬)$	0.00479	0.00019	0.00268	0.00019
$p(k \mid 猫)$	0.00083	0.00103	0.00186	0.00021
$\dfrac{p(k \mid 犬)}{p(k \mid 猫)}$	5.803	0.186	1.444	0.929

表 3.6: 単語 k が周辺に現れる確率とその比. 「飼育」や「彼女」はそれぞれ「犬」および「猫」の周辺に現れる確率が高く, 比は 1 より大または小となりますが, 両方に関係する「動物」およびどちらにも関係しない「本陣」では, 確率の比はほぼ 1 になります.

か. 式(3.107)でみたように

$$\mathrm{PMI}\,(w,c) = \log \frac{p(w,c)}{p(w)\,p(c)} = \log \frac{p(c|w)}{p(c)} \tag{3.120}$$

ですから, 両辺を指数の肩にのせれば,

$$\exp(\mathrm{PMI}\,(w,c)) = \frac{p(c|w)}{p(c)} \tag{3.121}$$

$$\therefore \quad p(c|w) = p(c) \cdot \exp(\mathrm{PMI}\,(w,c)) \tag{3.122}$$

となります. 式(3.122)は, 単語 w の周りに文脈語 c が現れる条件つき確率 $p(c|w)$ は, c のデフォルトの出現確率 $p(c)$ に, 係数 $e^{\mathrm{PMI}\,(w,c)}$ をかけたものになることを表しています. $\mathrm{PMI}\,(w,c)=0$, すなわち式(3.108)より $p(w,c)=p(w)p(c)$ で w と c が無相関の場合は, c の条件つき確率は $p(c)\cdot e^0=p(c)\cdot 1=p(c)$ でデフォルトの確率に等しく, $\mathrm{PMI}\,(w,c)>0$ なら確率は $e^{\mathrm{PMI}\,(w,c)}>1$ なので $p(c)$ より大きく, $\mathrm{PMI}\,(w,c)<0$ なら確率は $e^{\mathrm{PMI}\,(w,c)}<1$ なので $p(c)$ より小さくなるわけです. このように, 自己相互情報量は確率を式(3.122)のように「変調」する係数を表しており, これは統計学では Cox 比例ハザード[89]とよばれるモデルと同じモデルだといえます.

3.5.5* GloVe と意味方向の数理

こうして得られた単語ベクトルの間には, 3.5.1 節で示した「引き算」の関係

が成り立つことが知られていますが, これはなぜなのでしょうか. スタンフォード大学で素粒子物理を専攻していた Pennington は 2014 年に, こうした意味的関係が成り立つように設計された単語ベクトル, GloVe (Global Vector) [90]を提案しました. GloVe の単語ベクトルは Word2Vec の単語ベクトルと似ていますが, 目的関数が少し異なり, 若干違うベクトルになっています.

単語ベクトルについて要求される条件は, 「意味が似ているベクトルは値が近い」ということです. これを, 確率の言葉で表現するとどうなるでしょうか.

まず, ある単語 w の周囲に単語 c が現れる確率 $p(c|w)$ は, 前節で用いた窓内の共起頻度 $n(w,c)$ を使って,

$$p(c|w) = \frac{n(w,c)}{n(w)} \tag{3.123}$$

で計算できることに注意しましょう. いま, 2 つの単語 $i =$“犬” と $j =$“猫” があったとき, 別の単語 k がこれらの単語の周囲に現れる確率の比

$$\frac{p(k|i)}{p(k|j)} \tag{3.124}$$

をさまざまな k について計算すると, 表 3.6 のようになります. これから, 図 3.34 に示したように

- 犬に強く関係する単語 k, たとえば “飼育” は $p(k|$ 犬$)$ の方が $p(k|$ 猫$)$ よりも大きく,
$$\frac{p(\text{飼育} \mid \text{犬})}{p(\text{飼育} \mid \text{猫})} = 5.803 \ \gg 1$$
のように, 比が 1 よりもずっと大きくなります.
- 逆に, 猫に強く関係する単語 k, たとえば “彼女” は $p(k|$ 猫$)$ の方が $p(k|$ 犬$)$ より大きく,
$$\frac{p(\text{彼女} \mid \text{犬})}{p(\text{彼女} \mid \text{猫})} = 0.186 \ \ll 1$$
のように, 比が 1 よりずっと小さくなっています.
- 一方, 犬とも猫とも関係のある単語 “動物”, および関係のない単語 “本陣” は,

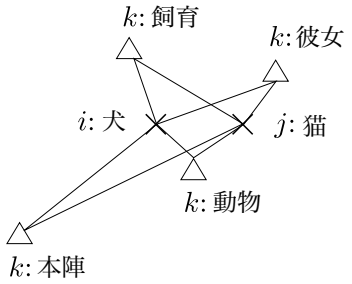


図 3.34: GloVe の用いる単語ベクトルの間の関係. それぞれの単語 k が単語 i, j の周辺に現れる確率 $p(k|i)$, $p(k|j)$ の比 $p(k|i)/p(k|j)$ から, 単語ベクトルを求めます.

$$\frac{p(\text{動物} \mid \text{犬})}{p(\text{動物} \mid \text{猫})} = 1.444 \ \approx 1$$

$$\frac{p(\text{本陣} \mid \text{犬})}{p(\text{本陣} \mid \text{猫})} = 0.929 \ \approx 1$$

と, どちらも 1 に近くなっています (ただし, 表 3.6 にみるように分子・分母の確率の絶対値は関係があれば大きく, なければ小さくなります).

つまり一般に, 図 3.34 のような単語の間の意味的関係は

$$\frac{p(k|i)}{p(k|j)} \equiv \frac{p_{ik}}{p_{jk}} \tag{3.125}$$

の値で示される, ということがわかります. ここで簡単のため, $p(k|i)=p_{ik}$, $p(k|j)=p_{jk}$ と表記しました. 式(3.125)は確率の比, すなわち相対値を見ているため, 個々の確率の絶対値 p_{ik}, p_{jk} が単語の頻度によって変わっても不変な値であることに注意してください. したがって, これから求める単語ベクトル $\overrightarrow{w_i}, \overrightarrow{w_j}, \overrightarrow{w_k}$ から式(3.125)が計算されるべきですから, その関数を一般に F とおけば,

$$F(\overrightarrow{w_i}, \overrightarrow{w_j}, \overrightarrow{w_k}) = \frac{p_{ik}}{p_{jk}} \tag{3.126}$$

が計算できて図 3.34 の関係が成り立つべきだ, ということになります.

関数 F にはいろいろな可能性がありますが, 3.5.2 節で示した「ベクトルの引き算」が成り立つためには, 最も単純には F は差 $\overrightarrow{w_i} - \overrightarrow{w_j}$ に依存する関数と

なるべきでしょう．また，式(3.126)の右辺はスカラー値ですから，線形な構造を保存してベクトルの次元の間に無用な相関を持ち込まないためには，$\overrightarrow{w_i} - \overrightarrow{w_j}$ と $\overrightarrow{w_k}$ の内積をとればよいと考えられます．よって，式(3.126)は

$$F\left((\overrightarrow{w_i} - \overrightarrow{w_j})^T \overrightarrow{w_k}\right) = \frac{p_{ik}}{p_{jk}} \tag{3.127}$$

と書き換えることができました．

ここで，式(3.127)の右辺は確率の比のため，必ず正であることに注意してください．よって F は必ず正の値を返す必要がありますが，左辺の ( ) の中は $\overrightarrow{w_i}^T \overrightarrow{w_k} - \overrightarrow{w_j}^T \overrightarrow{w_k}$ と差になっており，負になる可能性がありますから，一般に実数の和や差を正の数の積や割り算に変換できる F を考えれば [*84]，式(3.127)はさらに

$$F\left((\overrightarrow{w_i} - \overrightarrow{w_j})^T \overrightarrow{w_k}\right) = \frac{F\left(\overrightarrow{w_i}^T \overrightarrow{w_k}\right)}{F\left(\overrightarrow{w_j}^T \overrightarrow{w_k}\right)} = \frac{p_{ik}}{p_{jk}} \tag{3.128}$$

と書き換えることができます．この式を満たす解として $F(x) = \exp(x)$ があります．したがって

$$\exp\left(\overrightarrow{w_i}^T \overrightarrow{w_k}\right) = p_{ik} \tag{3.129}$$

すなわち

$$\overrightarrow{w_i}^T \overrightarrow{w_k} = \log p_{ik} = \log p(k|i) \tag{3.130}$$

が成り立てばよい，ということがわかります．頻度を使って書き換えれば，

$$\overrightarrow{w_i}^T \overrightarrow{w_k} = \log n(i,k) - \log n(i) \tag{3.131}$$

となります．式(3.130)が，これまでに述べた意味的な関係から単語ベクトルが満たすべき十分条件ということになります．

ここで，式(3.130)の左辺は i と k を入れ替えても成り立ちますが，右辺は条件つき確率なので，i と k を逆にすると意味が変わってしまうことに注意してください．そこで $\log n()$ に対応するバイアス項 b_i を，対称性から同様に b_k を導

*84　原論文では，F として群 $(\mathbb{R}, +)$ から $(\mathbb{R}_{>0}, \times)$ への準同型写像をとると説明されています．

入すれば,

$$\overrightarrow{w_i}^T \overrightarrow{w_k} = \log n(i,k) - b_i - b_k \tag{3.132}$$
$$\therefore \quad \overrightarrow{w_i}^T \overrightarrow{w_k} + b_i + b_k = \log n(i,k)$$

となり, i と k について対称な目的関数を得ることができました [*85].

ただし, 単純に式(3.132)の回帰モデルを求めるには問題があり, $n(i,k)=0$ の場合や, $n(i,k)>0$ でも, 偶然共起した場合をすべて説明しなければならなくなってしまいます. そこで, 式(3.132)の両辺の二乗誤差を, $n(i,k)$ に依存する関数 $f(n(i,k))$ で重みづけて, GloVe では

$$J = \sum_{i,k=1}^{V} f(n(i,k)) \Big(\overrightarrow{w_i}^T \overrightarrow{w_k} + b_i + b_k - \log n(i,k) \Big)^2 \tag{3.133}$$

を最小化します. $f(x)$ は $x=0$ のとき 0 で $x>0$ のとき少しずつ大きくなり, $x_{\max}$ で頭打ちになる関数で (GloVe では $x_{\max}=100$ を用いています),

$$f(x) = \begin{cases} \left(\dfrac{x}{x_{\max}}\right)^{\alpha} & x < x_{\max} \\ 1 & \text{それ以外} \end{cases} \tag{3.134}$$

という, 図 3.35 のような関数を用いると性能が良かったことが報告されていま

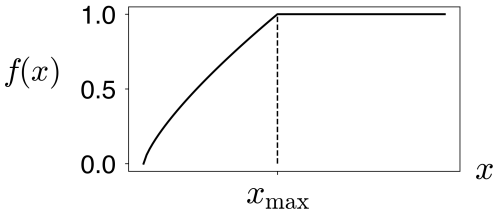
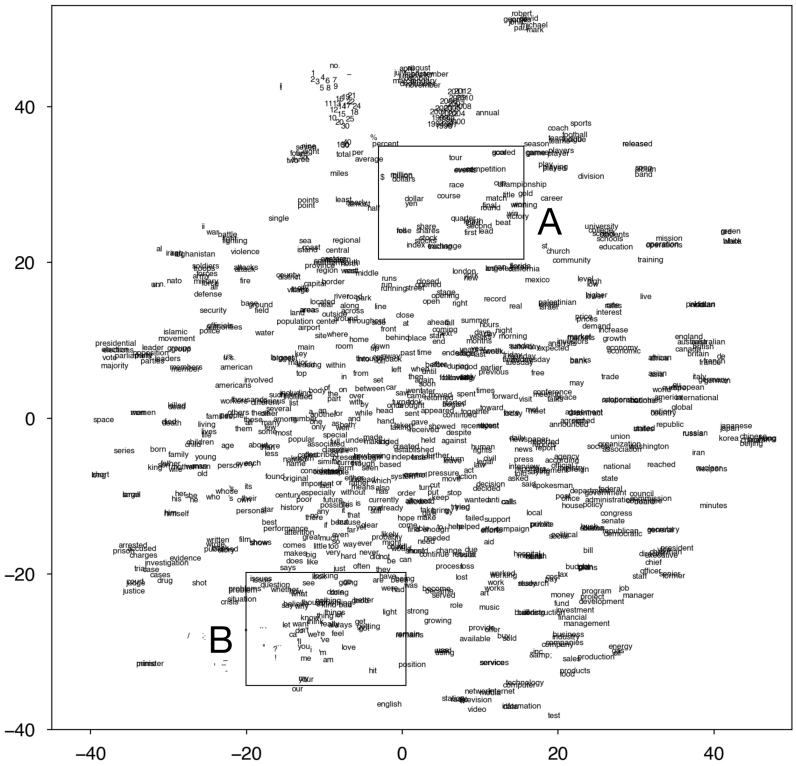
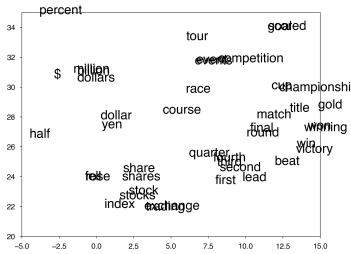


図 3.35: GloVe で用いている頻度 x の重みづけ関数 $f(x)$. $x=0$ では重み 0 に, $x_{\max}$ 以上ではすべて重みが 1 になります.

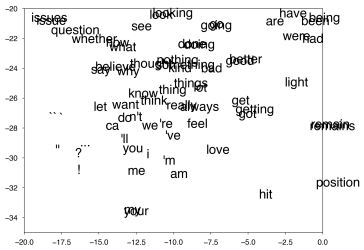
*85 GloVe の真の目的関数は式(3.130)ですが, 最適化のために導入しているこうしたヒューリスティックが, GloVe の性能を本来のものより下げている可能性があります. 直接, 式(3.130)を MCMC 法などで最適化することを考えてみてもよいでしょう.



(a) 全体図．数字や年号, 人名のクラスタや経済用語の集まっている場所などがあります．



(b) A を拡大したもの



(c) B を拡大したもの

図 3.36: GloVe で 6 億語のテキストから学習された頻度上位 1000 語の単語ベクトルの可視化．100 次元空間の単語ベクトルを, t-SNE で 2 次元に可視化しています．

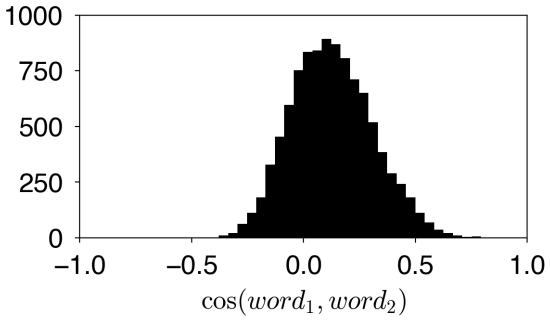


図 3.37: `ja.text8` コーパスから計算した単語ベクトルでの, ランダムな 2 単語間のコサイン類似度の分布. 類似度は $[-1, 1]$ ではなく, その一部にだけ分布しています.

す. なお, 本来は $\overrightarrow{w_i}$ と $\widetilde{w_k}$ の役割は同じですので, 最終的な単語ベクトルとしては平均をとった $(\overrightarrow{w_i}+\widetilde{w_k})/2$ が採用されています.

GloVe を使っても, `word2vec` とほとんど同様の単語類似度や比例関係を計算することができます (→演習問題 7). GloVe では, スタンフォード大学の公式ページで, Wikipedia 全体のような巨大なテキストから計算した英語の単語ベクトルが公開されています [*86]. このうち, 100 次元の単語ベクトルの頻度上位 1000 語を, t-SNE とよばれる非線形な可視化法で 2 次元に可視化したものを図 3.36 に示しました [*87].

3.5.6 単語ベクトルの分布とノルム

ここまで単語ベクトルについてみてきましたが, それでは, 学習された単語ベクトルは空間上でどのように分布しているのでしょうか. 以下では, Word2Vec を使って日本語の `ja.text8` コーパスから学習された単語ベクトルについて分析していくことにします.

単語ベクトル $\vec{w}$ と $\vec{v}$ の類似度は, 138 ページでみたようにそのなす角のコサインとして,

---

*86 `https://nlp.stanford.edu/projects/glove/`からダウンロードすることができます.

*87 この単語ベクトルの可視化は, サポートサイトの `visualize.py` を使うと行うことができます. 詳しくは, スクリプトの中身をご覧ください.

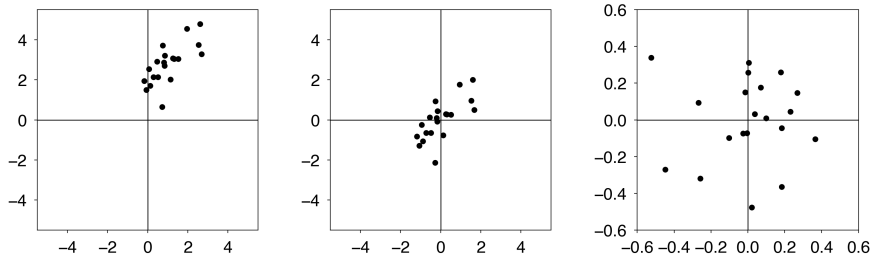
$$\cos(\vec{w}, \vec{v}) = \frac{\vec{w} \cdot \vec{v}}{|\vec{w}||\vec{v}|} \tag{3.135}$$

で測ることができます．任意の実数 x について $-1 \leq \cos x \leq 1$ ですから，式(3.135)は -1 から 1 の範囲に分布しているはずです．しかし，次のようにランダムな 2 単語について式(3.135) を計算してみると，図 3.37 のように最大値も最小値もまったく ± 1 ではなく，平均値も 0 ではないことがわかります．

```
from numpy.random import randint
from pylab import *
vectors = vload("model/ja.text8.vec")
words = list(vectors.keys())
N = 10000
V = len(words)
s = np.zeros (N, dtype=float)
for n in range(N):
    i = randint(V); j = randint(V)
    s[n] = similar(vectors, words[i], words[j])
hist(s,bins=30)
axis([-1,1,0,1000])
```

つまり，単語ベクトルは K 次元空間 (ここでは 100 次元空間) 上に，かなり偏って分布しているということです．実際，単語ベクトルの中心を次のようにして計算してみると，まったく 0 ではないことがわかります．

```
def loadvec (file):
    matrix = []
    with open (file, 'r') as fh:
        for line in fh:
            tokens = line.rstrip('\n').split()
            if len(tokens) > 2: # Word2Vec のヘッダをスキップ
                matrix.append (np.array (list ( \
                                map (float, tokens[1:]))))
    return np.array(matrix)
matrix = loadvec("model/ja.text8.vec")
np.mean(matrix, 0)
⇒ array([ 0.008, -0.003, -0.051, -0.025,  0.133,  0.147, -0.057,
          0.009,  0.126,  0.163,  0.027, -0.017, -0.037, -0.072, ...
         -0.107,  0.046, -0.170,  0.072])
```



(a) 元のデータ (b) 中心化を行ったもの (c) 白色化を行ったもの

図 3.38: ベクトルの中心化と白色化. 中心化により平均が 0 に, さらに白色化により共分散が単位行列 $\mathbf{I}$ になります.

単語ベクトルの中心化 よって, 最も自然に考えられる前処理は, 単語ベクトルから上の中心を引いて, 平均を 0 にすることでしょう (図 3.38(b)).

$$\vec{w}' = \vec{w} - \mu \quad ; \ \mu = \frac{1}{V} \sum_{v=1}^{V} \vec{v} \tag{3.136}$$

これは, データ解析では一般に, ベクトルの**中心化** (centering) とよばれています.

ただし実は, 式(3.136)のようなナイーブな中心化をしてしまうと, 単語ベクトルの表現力はむしろ悪くなってしまうことが知られています. 3.2 節で議論したように, 言語には大量の単語があり, そのほとんどはきわめて低確率なのでした. 単純な平均をとると, 「日本」「人権」のような頻度の高い重要な単語ベクトルと, 「鼠小僧次郎吉」「ザンペリーニ」のようなめったに出てこない特殊な単語ベクトルの平均をとることになってしまいます.

この問題に対し, 横井ら[91]は, 一様ではなく単語の確率で期待値をとって平均を計算することで, 単語ベクトルの性能が改善することを示しました. すなわち式(3.136)の代わりに,

$$\vec{w}' = \vec{w} - \mu \quad ; \ \mu = \sum_{v=1}^{V} p(v) \vec{v} \tag{3.137}$$

として単語ベクトルを中心化します. $p(v)$ は, 単語 v の出現確率です. サポートサイトに, この前処理を行うスクリプト `vcenter.py` を示しました. この処理に

は単語のユニグラム確率が必要ですので, 実行には

```
% vcenter.py ja.text8.vec ja.text8 output.vec
```

のように, 単語ベクトルの学習に使ったコーパスを指定します. なお, 単語ベクトルの学習によく使われる text8 や日本語版の ja.text8 には改行がなく, 巨大な 1 行のテキストになっているため, 2.2 節のような方法でテキストを行ごとに読み出すことはできません. こうした場合は, Python の yield を使って

```
import re

def readword (fh, newline=r'[ \t\n]+'):
    buf = ""    # 単語の読み出しバッファ
    while True: # ファイルが終わるまで
        while True: # バッファから単語を読み出す
            match = re.search (newline, buf)
            if not match:
                break
            else:
                yield buf[:match.start()]
                buf = buf[match.end():]
        chunk = fh.read (4096) # ファイルを一定量読む
        if not chunk: # ファイルの終了
            if len(buf) > 0:
                yield buf
            break
        buf += chunk  # バッファに読んだ分を追加
```

のようなジェネレータを定義すれば,

```
with open('ja.text8', 'r') as fh:
    for word in readword(fh):
        ...
```

のように単語を次々と読み出すことができます. 上の vcenter.py は, こうして単語のユニグラム確率を計算しています.

単語ベクトルの白色化　単語ベクトルの平均は中心化によって (期待値の意味で) 0 になりますが, 単語ベクトルの空間は図 3.38(b) のように, それでもまだ歪みを残しているはずです. こうした場合, データ解析でよく行われる前処理が **白色化** (whitening) です. 白色化は図 3.38(b) を図 3.38(c) のように線形射影によって変換する処理で, 平均を 0 にした上で, さらに異なる次元間の共分散を 0

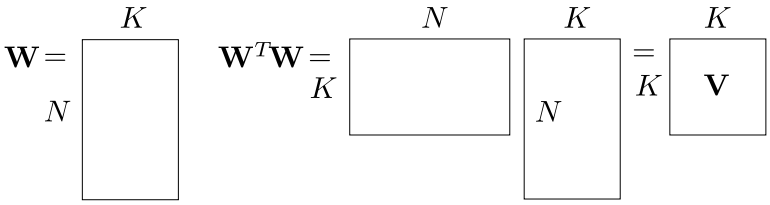


図 3.39: 単語ベクトルの行列 $\mathbf{W}$ とその共分散行列 $\mathbf{V}=\mathbf{W}^T\mathbf{W}$ の計算.

にして共分散行列を単位行列にします. つまり, この処理により K 次元空間の次元が「無駄なく」使われるようになります.

ベクトルの白色化は, 一般に次のようにして行うことができます. いま, 各データが K 次元の行ベクトルで, N 個のデータが図 3.39 左のように, $N\times K$ の行列 $\mathbf{W}$ をなしているとします [*88]. $\mathbf{W}$ を中心化して行方向の平均を $\mathbb{E}[\mathbf{W}]=\mathbf{0}$ にしたとき, K 次元の各次元間の共分散行列は, 図 3.39 右のように $\mathbf{V}=\mathbf{W}^T\mathbf{W}$ で計算することができます. $(\mathbf{W}^T\mathbf{W})^T=\mathbf{W}^T\mathbf{W}$ より $\mathbf{V}$ は対称行列ですから, $\mathbf{V}$ は

$$\mathbf{V}=\mathbf{P}\boldsymbol{\Sigma}\mathbf{P}^{-1} \tag{3.138}$$

と対角化することができます. ここで $\boldsymbol{\Sigma}$ は $\mathbf{V}$ の固有値 $\lambda_1,\cdots,\lambda_K$ を対角成分にもつ対角行列, $\mathbf{P}$ は対応する固有 (行) ベクトルを列方向に並べた行列です. したがって, $\mathbf{P}^T\mathbf{P}=\mathbf{I}$ よって $\mathbf{P}^T=\mathbf{P}^{-1}$ になります. このとき,

$$\mathbf{Z}=\mathbf{W}\mathbf{P}\boldsymbol{\Sigma}^{-1/2} \tag{3.139}$$

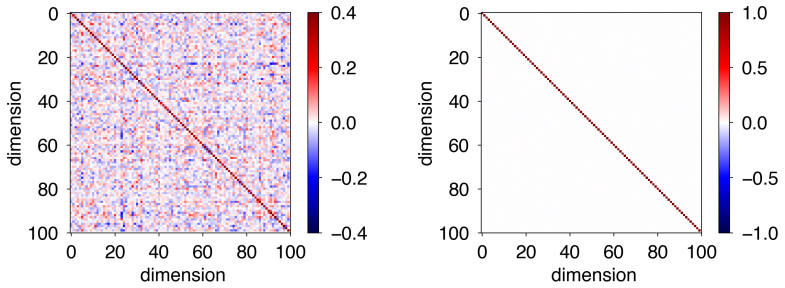
と $\mathbf{W}$ を $\mathbf{Z}$ に変換すれば [*89], その共分散は式(3.138)から,

$$\begin{aligned}\mathbf{Z}^T\mathbf{Z}&=(\mathbf{W}\mathbf{P}\boldsymbol{\Sigma}^{-1/2})^T\mathbf{W}\mathbf{P}\boldsymbol{\Sigma}^{-1/2}\\&=\boldsymbol{\Sigma}^{-1/2}\mathbf{P}^T\mathbf{W}^T\mathbf{W}\mathbf{P}\boldsymbol{\Sigma}^{-1/2}=\boldsymbol{\Sigma}^{-1/2}\underbrace{\mathbf{P}^T(\mathbf{P}}_{\mathbf{I}}\boldsymbol{\Sigma}\underbrace{\mathbf{P}^{-1})\mathbf{P}}_{\mathbf{I}}\boldsymbol{\Sigma}^{-1/2}\\&=\boldsymbol{\Sigma}^{-1/2}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1/2}=\mathbf{I}\end{aligned} \tag{3.140}$$

になります. すなわち, $\mathbf{W}$ を式(3.139)で線形変換した $\mathbf{Z}$ は平均 $\mathbb{E}[\mathbf{Z}]=\mathbf{0}$, 共

*88 数学ではベクトルは列ベクトルが基本ですが, NumPy の行列は標準ではデータを行方向に格納するため (row major といいます), あえてこの表現を使っています.

*89 $\boldsymbol{\Sigma}^{-1/2}$ は, $1/\sqrt{\lambda_1},\cdots,1/\sqrt{\lambda_K}$ を対角成分にもつ対角行列です.



(a) もとの単語ベクトルの共分散行列　(b) 白色化後の単語ベクトルの共分散行列

図 3.40: 単語ベクトルの共分散行列と白色化．白色化により，単語ベクトルの各次元を無相関にすることができます．

分散 $\mathbb{V}[\mathbf{Z}]=\mathbf{I}$ で分布するようになります．これを**白色化**といいます．□

したがって，単語ベクトルを並べた行列 $\mathbf{W}$ についてこの白色化を行えば，図 3.38(c) のようにデータは K 次元空間で無駄なく分布するようになり，細かい意味の差を反映するようになると考えられます．ただし，ここでも行列 $\mathbf{W}$ の各行に対応する単語には大きな頻度の差があるため，[91] では式 (3.137) の期待値を用いた中心化を行った上で，$\mathbf{W}$ の各行 $\overrightarrow{w_i}$ を，長さを 1 に規格化した上で $\sqrt{p(w_i)}$ で重みづけた

$$\overrightarrow{w_i}' = \sqrt{p(w_i)}\frac{\overrightarrow{w_i}}{|\overrightarrow{w_i}|} \tag{3.141}$$

としてから白色化を行うことで，単語ベクトルの性能が向上することを示しています．白色化の処理は同様ですので，結果として新しい単語ベクトルはやはり平均 $\mathbf{0}$，共分散 $\mathbf{I}$ で分布することになります．この処理を，期待白色化とよびましょう．

サポートサイトに，この期待白色化を行うスクリプト `vwhiten.py` を示しました．

```
% vwhiten.py ja.text8.vec ja.text8 whitened.vec
```

を実行すると，`whitened.vec` に期待白色化された単語ベクトルが保存されます．

図 3.40 に，もとの単語ベクトルの共分散行列 $\mathbf{V}$ と，期待白色化後の共分散行列 $\mathbf{V}$ を示しました．もとの単語ベクトルでは各次元の間に ± の多くの相関が

ありますが, 変換後はそれが一掃され, 単語ベクトルの各次元はすべて無相関になっている (共分散が単位行列になっている) ことがわかります.

この新しい単語ベクトルで単語の類似度を計算してみると, 下のようになります. 140 ページの結果と比べてみると, “太陽” の類似語の上位から “シリウス” や “星雲” が外れていたり, “少年” の類似語の順番が入れ替わっていたりなど, 微妙にではありますが, 確実に類似度が改善していることがわかります. 数値的にも, 単語類似度や特に文類似度などのタスクの性能で改善がみられることが報告されています[91].

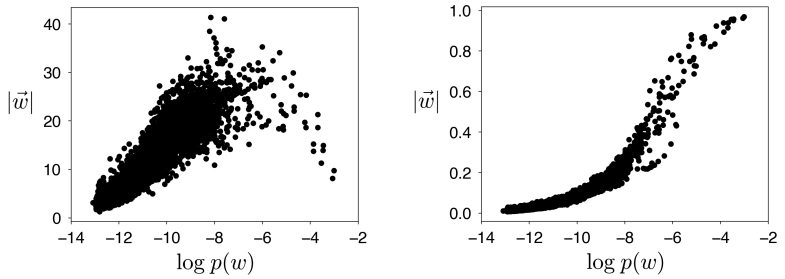
```
  similars(whitened, "太陽")
⇒ 恒星 -> 0.7570
  太陽系 -> 0.7068
  星 -> 0.6908
  土星 -> 0.6870
  地球 -> 0.6804
  惑星 -> 0.6752
  質量 -> 0.6711
  天体 -> 0.6626
  海王星 -> 0.6590
  銀河 -> 0.6485
  超新星 -> 0.6480
```

```
  similars(whitened, "少年")
⇒ 少女 -> 0.8279
  小学生 -> 0.6740
  中学生 -> 0.6734
  高校生 -> 0.6566
  青年 -> 0.5830
  同級生 -> 0.5791
  幼児 -> 0.5754
  忍者 -> 0.5253
  中学 -> 0.5179
  大人 -> 0.4748
  成人 -> 0.4667
```

単語ベクトルの長さ これまでは, 単語ベクトルの類似度や比例関係はすべて, 長さを 1 に規格化した上で行ってきました. しかし実際は, 単語ベクトルはその方向だけでなく, 長さの情報も持っています. 単語ベクトルの長さは, どうなっているのでしょうか.

単語ベクトルの長さ $|\overrightarrow{w}|$ を縦軸に, その単語の出現確率 $p(w)$ の対数を横軸にとってプロットしてみると, 図 3.41 のように, 確率が中程度の単語ベクトルの長さが最も大きいことが知られています[92]. 確率的勾配法で単語ベクトルを最適化するとき, 頻度の高い語のベクトルは様々な方向から「引っ張られる」ため, 結果的に長さは短くなるでしょう. また頻度の低い語は逆にわずかしか引っ張られませんので, 単語ベクトルは短いままになると考えられます.

ただし, こうした頻度の影響を差し引いた上では, 意味が「強い」単語のベクトルはノルムが大きい傾向にあることが報告されています[93]. ここで単語 w の意味が「強い」とは, w の周囲に出現する単語 c の確率分布 $\{p(c|w)\}$ と, 単



(a) Word2Vec での単語ベクトルの場合　(b) 期待白色化を行った後の関係.

図 3.41: 単語の確率 (横軸) と単語ベクトルの長さ (縦軸) の関係.

語 c の平均的な確率分布 $\{p(c)\}$ との KL ダイバージェンス (式(2.69)) が大きい, すなわち周辺語の確率分布が偏っていることを意味します.

なお, 単語ベクトルの期待白色化を行うと, 頻度と長さの関係は図 3.41(b) のように単純な比例関係に変わります. Word2Vec だけでなく, GloVe や PMI の場合にどうなるかなど, 単語ベクトルの長さについては統一的な理論が待たれています. (→演習 (7))

3 章の演習問題

(1) 実際のテキストで, Heaps の法則の γ を線形回帰で求めてみる. 線形回帰の係数の求め方については, GP 本の第 1 章を参照のこと.

(2) 単語と異なり, 文字は言語では一般に有限です. 有限集合である文字の場合にテキストを読みながら文字の「語彙」の大きさを計算した場合, Heaps の法則は成り立つでしょうか? 文字の種類の少ない英語と多い日本語では, 違いがあるでしょうか?

(3) Zipf の法則の α は, 実際のテキストではどの程度の値になるでしょうか. 図 3.4 の曲線を, 1 つの直線で完全に表すことはできるでしょうか.

(4) 部分積分により, $\Gamma(x+1) = x\Gamma(x)$ を示せ.

(5) 3.5.6 節節では, `ja.text8` について Word2Vec で学習された日本語の単語ベクトルについて考えてみましたが, GloVe や PMI で学習されたベクトルについては, 分布はどうなっているでしょうか. Word2Vec との間に, 何か違いがあるでしょうか?
また, 英語の単語ベクトルについてはどうでしょうか. 日本語の単語ベクトルの場合と, 何か違いがあるでしょうか.

(6) 単語ベクトルの次元を変えたとき, 結果はどう変わるでしょうか. タスクについて.

(7) GloVe を使って同様に単語類似度や, 単語の比例関係を計算してみる. C 言語の実装は公式ページにあります. 同じテキストから学習したとき, Word2Vec と何か違いがあるでしょうか.

(8) A la Carte embedding について.