2　文字の統計モデル

2.1　文字の頻度と出現確率

それでは, 実際にテキストをみていくことにしましょう. どんなテキストも, 改行を含めて文字の系列, すなわち**文字列**とみなすことができます. 日本語や中国語のような言語は空白で単語に分けられていませんので, 文字列は最も基本的で重要な表現です. 英語のほうが日本語より文字の種類が少ないので, 簡単のために, まずは英語の例で考えてみることにしましょう.

本書のサポートサイト (「はじめに」v ページ) にある『不思議の国のアリス』のテキスト `alice.txt` は 1,430 行, 長さ 132,656 文字のテキストで, 図 2.1 のような内容になっています.[*1] 説明のため, すべて小文字にして記号を削除してありますので[*2], 文字は “a”, …, “z” およびスペース “␣” の 27 文字からなっています.

このファイルを文字列として Python に読み込むには, 次のように実行します.

```
with open ('alice.txt', 'r') as f:
    s = f.read()
```

これで, `alice.txt` の内容が文字列 `s` に代入されました. `s` の中身を見てみると,

```
s
⇒ 'alices adventures in wonderland\nlewis carroll\nchapter i
  \ndown the rabbithole\nalice was beginning to get very tir
```

---

*1　このテキストは, 著作権の切れた文学作品を集めた Project Gutenberg (`https://www.gutenberg.org/`) から入手したものです.

*2　英語の場合は何を大文字にするかには規則性がありますので, すべて小文字にしても情報量が落ちることはあまりなく, 機械学習を適切に使えば, 原文をほぼ復元することができます.

```
alices adventures in wonderland
lewis carroll
chapter i
down the rabbithole
alice was beginning to get very tired of sitting by her
sister on the bank and of having nothing to do once or twice
she had peeped into the book her sister was reading but it
had no pictures or conversations in it and what is the use
of a book thought alice without pictures or conversation
so she was considering in her own mind as well as she could
for the hot day made her feel very sleepy and stupid whether
the pleasure of making a daisychain would be worth the
trouble of getting up and picking the daisies when suddenly
a white rabbit with pink eyes ran close by her there was
nothing so very remarkable in that nor did alice think ...
```

図 2.1: 『不思議の国のアリス』 alice.txt の冒頭部.

```
ed of sitting by her sister on the bank and of having noth
ing to do once or twice she had ...'
```

のようになっています. “\n” は, 改行を表す特別な文字です [*3].

この中で, それぞれの文字が何回くらい現れるのか, 頻度を数えてみることにしましょう. 改行文字は無視することにすると,

```
from collections import defaultdict
freq = defaultdict (int)
for c in s:           # c = s[0],s[1],... を順に調べる
    if c != '\n':     # c が改行文字以外の場合
        freq[c] += 1 # 頻度を増やす
```

で文字の頻度を数えることができます. #以下はコメントで無視されますので, 入力する必要はありません. 頻度を数えるテーブルの freq は Python の辞書 (dict) なので, freq = {} として初期化してもよいのですが, freq[c] の初期値を自動的に 0 にするため, collections モジュールの defaultdict を使っています.

結果は, 次のようになりました.

```
for (c,n) in freq.items():
```

*3 バックスラッシュ\は特別な文字で, 続く 1 文字と合わせて 1 つの文字を表します. 本書では, 実行例で行が続くことも\で表します.

```
    print ('%s = %d' % (c,n))
⇒ a = 8791
  l = 4713
  i = 7510
  c = 2398
  e = 13571
  s = 6500
  ␣ = 24966 ...
```

頻度順になっていないので, 少し見づらいですね. 頻度順に表示するには, 次のように入力します.

```
for (c,n) in sorted (freq.items(), # 頻度の降順にソート
                     key=lambda x: x[1], reverse=True):
    print ('%s = %d' % (c,n))
⇒ ␣ = 24966
  e = 13571
  t = 10687
  a = 8791
  o = 8145
  i = 7510
  h = 7373
  n = 7013
  s = 6500
  ...
  q = 209
  x = 148
  j = 146
  z = 78
```

頻度の合計は, 最初に述べたテキストの長さ 132656 と等しくなります. これを N としましょう.

```
sum (freq.values())
⇒ 132656
```

これから, 各文字の出現確率を計算することができます. 文字 c の頻度を $n(c)$ (上の Python コードでは `freq[c]`) とおくと, c が出現する確率は, 最も単純には

$$p(c) = \frac{n(c)}{N} \tag{2.1}$$

と考えてよいでしょう. $p()$ は, 確率を表す記法です. [*4] たとえば, 上の例では $n(\mathtt{e})=13571$, $n(\mathtt{a})=8791$ でしたから, 文字 e や a の確率は

$$p(\mathtt{e}) = \frac{n(\mathtt{e})}{N} = \frac{13571}{132656} = 0.1023 \tag{2.2}$$
$$p(\mathtt{a}) = \frac{n(\mathtt{a})}{N} = \frac{8791}{132656} = 0.0663$$

と求めることができます. 一方, 頻度の小さい q や z の確率は

$$p(\mathtt{q}) = \frac{n(\mathtt{q})}{N} = \frac{209}{132656} = 0.0016 \tag{2.3}$$
$$p(\mathtt{z}) = \frac{n(\mathtt{z})}{N} = \frac{78}{132656} = 0.0006$$

のような値になります. だいたい 150 倍くらいの違いがありますね. Python で各文字 c の確率を計算して変数 p[c] に保存するには, 次のように実行します.

```
p = {}
N = sum (freq.values())
for (c,n) in freq.items():
    p[c] = n / N
print(p)
⇒ {'a': 0.06626914726812207,
 'l': 0.035527982149318536,
 'i': 0.05661259196719334,
 'c': 0.0180768302979134,
 'e': 0.10230219515136896, ...
}
```

図 2.2 に, こうして計算した『不思議の国のアリス』の記号を除く各文字の出現確率と, その棒グラフを示しました. 英語では a, e, i, o のような母音の確率が際立って高く, 子音でも r, s, t などは確率が高いことがわかります. 一方で, j や q, x は非常に低い出現確率 (0.001 以下) になっています. 図 2.2 の各文字

*4 Pr() と書いたり, 離散のとき $P()$, 連続のとき $p()$ と分けて書く流儀もありますが, 混同する危険性はありませんから, 本書では機械学習の分野で最も標準的な $p()$ で表すことにします. $p(x)$ は一般に, "p of x" (probability of x の意味) と読みます. 筆者は個人的に, 「ピーエックス」や「p の x」と日本語で読んでしまうこともあります.

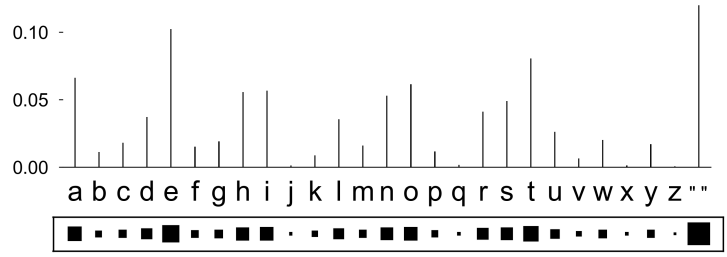


図 2.2:『不思議の国のアリス』`alice.txt` における各文字の出現確率.

の下にある■はヒントン図 (Hinton diagram)[*5] といい, ■が大きいほど確率が高いことを表しており, 確率の大きさを直感的に表現するためによく用いられます. 式(2.1)の確率は最尤(さいゆう)推定とよばれ, 現在手元にあるデータの確率 (これを 32 ページで説明するように, 尤度(ゆうど)といいます) を最大にする値ですが [*6], ここでは簡単に, 頻度を割り算して計算するもっとも簡単な確率の推定値だと考えてください.

図 2.2 のそれぞれの棒は文字の確率を表していますから, その長さの総和は必ず 1 になります. このように, 総和が 1 となる確率を並べたものを**確率分布**といいます. 図 2.2 は,『不思議の国のアリス』の裏に隠れた, 文字の確率分布です.

2.2　文字の同時確率

上では文字をばらばらに数えていましたが, 実際には英語では, “`es`” や “`li`” のような 2 文字の連続の確率が高く, “`qy`” や “`lg`” のような文字の連続の確率は非常に低いと考えられます. こうした 2 文字の連続を, **バイグラム** (bigram) といいます. これに対して, 2.1 節で考えた 1 文字を**ユニグラム** (unigram) といいます [*7]. 「グラム」とは,「文字」の意味です. バイグラムの出現確率は, ど

*5　ニューラルネットワークと深層学習の基礎を作ったことで有名なトロント大学の Geoffrey Hinton 教授によるもので, 本書のサポートサイトにも Python の実装 `hinton.py` が置いてあります.

*6　式(2.1)の確率が手元のデータの確率を最大にすることは, 数学的には自明ではなく, ラグランジュの未定乗数法を使った簡単な証明が必要です. 本章は導入のため証明は割愛していますので, 興味のある方は[7, 1.5.2 節]などを参照してください.

*7　古典ギリシャ語的な表現であるユニグラム, バイグラム, …の代わりに, それぞれ数字で 1 グラム, 2 グラム, …と言うこともあります.

うやって調べればよいでしょうか.

答えは簡単で, たとえば文字列が s = "alice" であれば, "al", "li", "ic", "ce"のように 1 文字ずつずらしながら, 2 文字の連続を同様に数えていけばよいだけです. この場合, s の最後の文字 “e” には続きの文字がありませんので, 後で説明する理由で, 文字列の最後に文字列の終わりを表す特別な文字 “$”[*8] があるとして計算すると, バイグラムの総数は文字列の長さと等しくなります. バイグラムの頻度は, Python では次のようにして数えることができるでしょう. ここから後は, テキストが非常に大きい場合にメモリを消費しないよう, はじめの例のようにファイルから文字列 s に一気に読み込むことはせず, 1 行ずつ読んでいくことにします. alice.txt では, 1 行がほぼ 1 文に対応しています.

```
freq = defaultdict (int)
with open ('alice.txt', 'r') as f:
    for line in f:             # 改行文字を含めて 1 行ずつ読む
        s = line.rstrip('\n') # 行の最後の改行文字を除去
        L = len(s)
        s += '$'               # 文字列末尾を表す特殊文字を追加
        for i in range(L):     # 1 文字ずつずらして数える
            b = s[i:i+2]       # 文字バイグラム (2 文字)
            freq[b] += 1       # 頻度を増やす
```

数えた結果は, 以下のようになりました.

```
for (b,n) in sorted (freq.items(), # 降順に表示
                     key=lambda x: x[1], reverse=True):
    print ('%s = %d' % (b, n))
print ('total %d bigrams.' % sum(freq.values()))
```

```
⇒ e␣ = 5417
␣t = 4192
he = 3778
th = 3483
␣a = 3152
...
ju = 102
y$ = 102
od = 101
oc = 99
```

*8 この “$” は説明のための表記で, 実際には本当の$と区別するために, テキストに出現しない特別な文字を用います. 詳しくは, 36 ページの脚注を参照してください.

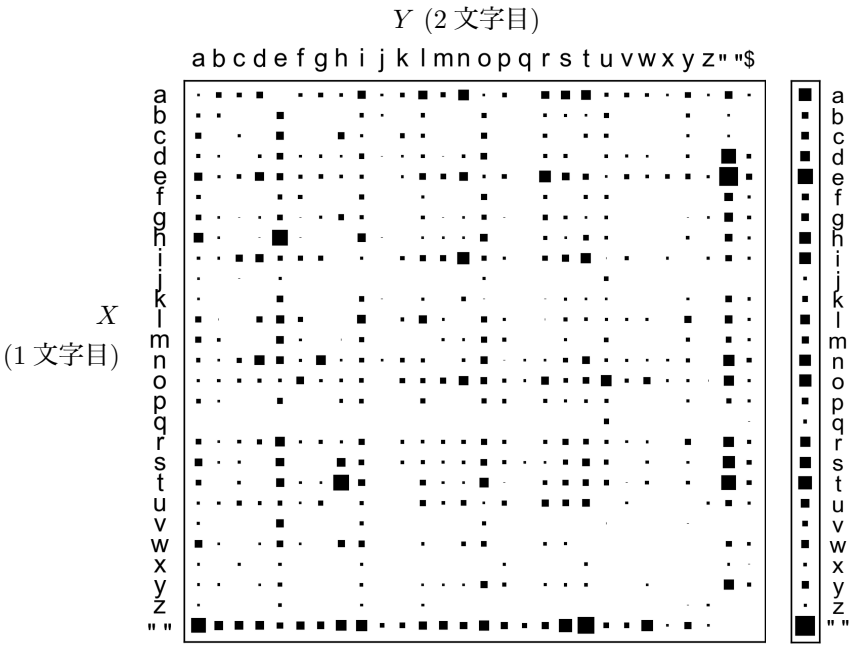


図 2.3: 『不思議の国のアリス』から計算した文字バイグラム確率 $p(X,Y)$ を表す行列 $\mathbf{P}$. 確率の総和は, 行列全体で 1 になっています. 右側に, 行列を横方向に和をとった, 各文字の周辺確率 $p(X)$ を示しました.

```
...
yy = 1
tv = 1
gb = 1
total 132656 bigrams.
```

こうすると, バイグラム b (たとえば b=`th`) の確率は式(2.1)と同様に,

$$p(b) = \frac{n(b)}{N} \tag{2.4}$$

で計算することができます. b=`th` の場合は,

$$p(\texttt{th}) = \frac{n(\texttt{th})}{N} = \frac{3483}{132656} = 0.0263 \tag{2.5}$$

となりました. Python では, バイグラムの確率は最初の文字ごとに辞書の形にして,

```
p = {}
N = sum (freq.values())
for (b,n) in freq.items():
    x = b[0]; y = b[1]
    if not(x in p):
        p[x] = {}    # 1 文字目ごとに別の辞書にする
    p[x][y] = n / N  # バイグラムの確率を保存
```

のように持つと, 計算が便利になるでしょう.

こうして計算したバイグラム確率を, バイグラムの 1 文字目を縦, 2 文字目を横にとって図 2.3 に示しました. バイグラムの文字には順番があるため, $p(\mathtt{qu})$ は $p(\mathtt{uq})$ とは等しくない, つまり図 2.3 の行列は対称ではないことに注意してください.

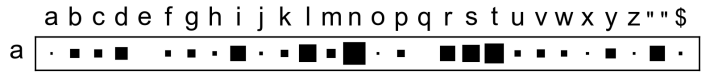
バイグラムの 1 文字目と 2 文字目を表す変数をそれぞれ X, Y とおくと, バイグラムの確率は

$$p(\mathtt{ab}) = p(X = \mathtt{a}, Y = \mathtt{b}) \tag{2.6}$$

のように表すことができます. 式(2.6)の右辺は「1 文字目 X が a」かつ「2 文字目 Y が b」という複数の条件が同時に満たされる確率を表していますので, こうした確率 $p(X, Y)$ を**同時確率**, あるいは**結合確率** (joint probability) といいます. ユニグラム確率 $p(X)$ は文字 X の生起確率ですが, バイグラム確率は 1 文字目 X と 2 文字目 Y の同時確率になっています. なお, こうした X, Y のような「役割」を表す変数のことを, **確率変数**といいます.

2.3 同時確率の周辺化

図 2.3 は, バイグラム確率 $p(X, Y)$ を表す行列でした. この行列の各行は, X をある文字に決めたときの $p(X, Y)$ の値になっています. たとえば, $X = \mathtt{a}$ の行だけを取り出すと



Y	a	b	c	d	e	f	g	h	…	$	合計
$p(X=\mathtt{a},Y)$	0.000	0.002	0.001	0.003	0.000	0.000	0.001	0.000	…	0.000	0.066

表 2.1: 同時確率 $p(X{=}\mathtt{a},Y)$ の表. 0.001 未満の確率は四捨五入されています.

のようになっていますが, これは

$$p(X{=}\mathtt{a},Y{=}\mathtt{a}),\ p(X{=}\mathtt{a},Y{=}\mathtt{b}),\ p(X{=}\mathtt{a},Y{=}\mathtt{c}),\ \cdots,\ p(X{=}\mathtt{a},Y{=}\mathtt{\$}) \tag{2.7}$$

を並べたもので, 確率としては表 2.1 のようになっています. [*9]

式(2.7) は, もとの X,Y を使わない表記では

$$p(\mathtt{aa}),\ p(\mathtt{ab}),\ p(\mathtt{ac}),\ p(\mathtt{ad}),\ \cdots,\ p(\mathtt{a\$}) \tag{2.8}$$

すなわち $p(\mathtt{a}*)$ ($*$ は任意の 1 文字) のことですから, これらの確率の総和は a が現れる確率, すなわち $p(\mathtt{a})$ と等しくなります. つまり, a 自体が現れる確率は, a の後に色々な文字が現れる場合の確率 $p(\mathtt{aa}),p(\mathtt{ab}),p(\mathtt{ac}),\cdots$ をすべて足したものになるわけです. 当たり前ですね. このことを X,Y を使って表すと,

$$\begin{aligned} p(X=\mathtt{a}) &= p(X=\mathtt{a},Y=\mathtt{a}) + p(X=\mathtt{a},Y=\mathtt{b}) + \cdots + p(X=\mathtt{a},Y=\mathtt{\$}) \\ &= \sum_Y p(X=\mathtt{a},Y) \end{aligned} \tag{2.9}$$

ということになります. 式(2.9)を, 同時確率 $p(X{=}\mathtt{a},Y)$ の**周辺化** (marginalization) といいます. これは図 2.3 で行列の各行の和をとる, すなわち行列を Y に関して横方向につぶして, 値を行列の「周辺」(margin, 縁) に集めていることに相当しますので, これが「周辺化」の名前の由来です. 同時確率の周辺化は, 一般的に

$$p(X) = \sum_Y p(X,Y) \qquad (\textbf{同時確率の周辺化の公式}) \tag{2.10}$$

と表すことができます. この公式は, X と Y が同時に現れる確率をすべての Y について和をとれば, X だけの確率になるという当たり前のことを表しています.

---

*9　Hinton 図は相対的な確率を示すものですので, ■が大きくても絶対的な確率が 1 に近いとは限らないことに注意してください.

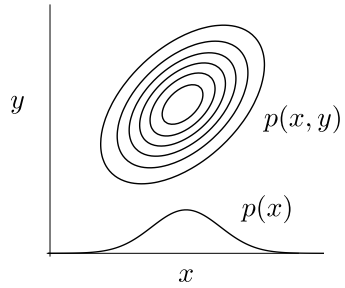


図 2.4: 同時確率密度 $p(x,y)$ の周辺化. y に関して積分して確率密度関数を下に「つぶす」ことで, x だけの関数 $\int p(x,y)\,dy = p(x)$ が得られます.

実際に確かめてみましょう. 上の例では,

$$p(\mathtt{aa}) + p(\mathtt{ab}) + \cdots + p(\mathtt{a\$}) = 0.000 + 0.002 + \cdots + 0.000 = 0.066 \tag{2.11}$$

ですが, 式(2.3)より $p(\mathtt{a}) = 0.066$ でしたので, この 2 つはぴったり同じになっています. Python では, バイグラムで計算した `p[x][y]` を使って

```
for x in p:
    s = 0
    for y in p[x]:
        s += p[x][y]
    print ('p(%s) = %f' % (x,s))
```

```
⇒ p[a] = 0.066269
  p[l] = 0.035528
  p[i] = 0.056613
  p[c] = 0.018077
  p[e] = 0.102302
  ...
```

とすれば, バイグラム確率を周辺化してユニグラム確率を計算することができます. 結果はもちろん図 2.2 と同じになり, それを図 2.3 の右側の縁に示しました.

同時確率の周辺化は言語のように離散値の場合だけでなく, 連続値の場合でも同様に成り立ちます. たとえば $p(x,y)$ が図 2.4 のように 2 次元のガウス分布

(正規分布) のとき, y 方向に分布をつぶして和をとれば, x の分布 $p(x)$ は 1 次元のガウス分布になることが知られています[6]. これは,

$$p(x) = \int p(x,y)\,dy \quad (\textbf{同時確率の周辺化の公式 (連続値の場合)}) \tag{2.12}$$

を計算していることになります. このように, $p(x,y)$ が確率密度の場合でも, 和を積分に置き換えれば, 式(2.12)の形で同時分布の周辺化が成り立ちます [*10].

◇

同時確率と周辺化の一般的な説明では, サイコロを 2 つ用意して, X を 1 個目のサイコロの目, Y を 2 個目のサイコロの目とすることが多いようです. このとき

$$p(X{=}1) = p(X{=}2) = \cdots = p(X{=}6) = \frac{1}{6} \tag{2.13}$$

ですが, 2 つのサイコロは独立なので, 特定の (X,Y) が出る確率はすべて

$$p(X{=}1,Y{=}1) = p(X{=}1,Y{=}2) = \cdots = p(X{=}6,Y{=}6) = \frac{1}{6}\times\frac{1}{6} = \frac{1}{36} \tag{2.14}$$

になります. このとき, たとえば 1 個目のサイコロの目が $X=1$ であれば, $p(X,Y)$ を 2 個目のサイコロの目 Y について周辺化すれば

$$\begin{aligned}\sum_Y p(X{=}1,Y) &= p(X{=}1,Y{=}1) + p(X{=}1,Y{=}2) + \cdots + p(X{=}6,Y{=}6) \\ &= \frac{1}{36} + \frac{1}{36} + \cdots + \frac{1}{36} = \frac{1}{6}\end{aligned} \tag{2.15}$$

となり,

$$\sum_Y p(X{=}1,Y) = p(X{=}1) \tag{2.16}$$

が成り立つことがわかります.

サイコロの例では確率 $p(X,Y)$ がすべて等しくなっていましたが, われわれ

---

*10　ライプニッツの積分記号 $\int$ は, そもそも Sum (和) を表す S の変形であることを思い出してください. 厳密にはこれらはすべて証明が必要ですが, 本書は確率論の本ではありませんので, 立ち入らないことにします. 基礎的な定義が気になる方は, [8]や最近の[9]といった優れた教科書を参照してください.

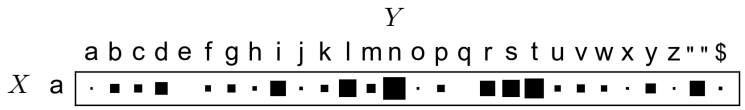
の言語の例のように $p(X, Y)$ がそれぞれ違う場合でも, 式(2.9)でみたように確率の周辺化は一般的に成り立ちます.

ここまでは行列の行について和をとる, つまりバイグラム確率 $p(\mathtt{a}*)$ の 2 文字目 $*$ について周辺化することで確率 $p(\mathtt{a})$ を求めてきましたが, 1 文字目について周辺化することも可能です. すなわちこれは, $p(*\mathtt{a})$ の形のバイグラム確率の和をとることになり, 図 2.3 の行列 $\mathbf{P}$ を列方向に周辺化することに相当します. われわれの例の場合は, X と Y の範囲が同じなので, これでも $p(\mathtt{a})$ を求めることができます. 計算して確かめてみてください. (→演習 1)

2.4 文字の条件つき確率

図 2.3 をよく見ると, q の行のうち $p(\mathtt{qu})$ だけが正で, あとはほとんど 0 になっていることがわかります. つまり, 1 文字目が q であれば, 2 文字目に u が来る確率はほとんど 1 だということです.[*11] それでは一般に, 文字 X の次に文字 Y が続く, すなわち「X が起きたときに Y が起きる確率」はどう計算すればよいのでしょうか. この確率を**条件つき確率**といい, $p(Y|X)$ と書きます. | の右側の X が条件, 左側の Y が結果で, 日本語の語順とは逆になっていることに注意してください.[*12]

2.3 節で取り出した, 『不思議の国のアリス』の文字バイグラムの同時確率の a の行



は, 「1 文字目の X が a で, かつ 2 文字目が Y である確率」を並べたものでした. よって, この中での**相対的な確率**を計算すれば, 「1 文字目が $X = \mathtt{a}$ だった場合の 2 文字目の確率」を求めることができるでしょう. たとえば, a の次に c が来る確率は

---

*11 alice.txt では実際に q の後には u しか来ませんので, この確率は式(2.1)のような最尤推定では 1 になります. 一般には Qatar (カタール) や QC (品質管理) のように別の文字が続く場合もありますが, その確率は u に比べるとごく小さいでしょう.

*12 $Y|X$ は一般に "Y given X" と読み, 英語ではこの語順通りになります. "given" は, 「〜が与えられたとき (= if)」という意味です.

(2.17)　$p(Y=\mathtt{c}|X=\mathtt{a})$

$$= \frac{p(X=\mathtt{a},Y=\mathtt{c})}{p(X=\mathtt{a},Y=\mathtt{a})+p(X=\mathtt{a},Y=\mathtt{b})+\cdots+p(X=\mathtt{a},Y=\mathtt{\$})}$$

$$= \frac{0.001}{0.000+0.002+\cdots+0.000} = \frac{0.001}{0.0066} = 0.015$$

と計算することができます.

2.3 節で, 式(2.17)の 2 行目の分母は同時確率の周辺化によって

$$p(X=\mathtt{a},Y=\mathtt{a})+p(X=\mathtt{a},Y=\mathtt{b})+\cdots+p(X=\mathtt{a},Y=\mathtt{\$}) = p(X=\mathtt{a})$$

であることをみました. よって, 式(2.17)は

(2.18)　$$p(Y=\mathtt{c}|X=\mathtt{a}) = \frac{p(X=\mathtt{a},Y=\mathtt{c})}{p(X=\mathtt{a})}$$

と表すことができます. すなわち一般に, 条件つき確率は

(2.19)　$$p(Y|X) = \frac{p(X,Y)}{p(X)}$$　(**条件つき確率の公式**)

で計算することができます. この式は文字通り, X が起きたときに Y が起きる条件つき確率 $p(Y|X)$ は, X が起きる確率 $p(X)$ に対する, Y も同時に起きる確率 $p(X,Y)$ の相対的な割合だという当たり前のことを表しています.

これを見ると, j の後に u が来る確率 $p(\mathtt{u}|\mathtt{j})$ が非常に大きいことや, w の後に k が来る確率 $p(\mathtt{k}|\mathtt{w})$ はほとんど 0 であることなど, 興味深い規則性をさまざまに読み取ることができるでしょう. (→演習 (3))

18 ページで計算した同時確率 q[x][y] を使うと, 式(2.19)の条件つき確率は次のようにして計算して c[x][y] に保存することができます.

```
p0 = {}; c = {}
chars = p.keys()
for c in chars:
    p0[c] = sum(p[c].values())          # 周辺確率の計算
for x in chars:
    c[x] = {}
    for y in chars:
```

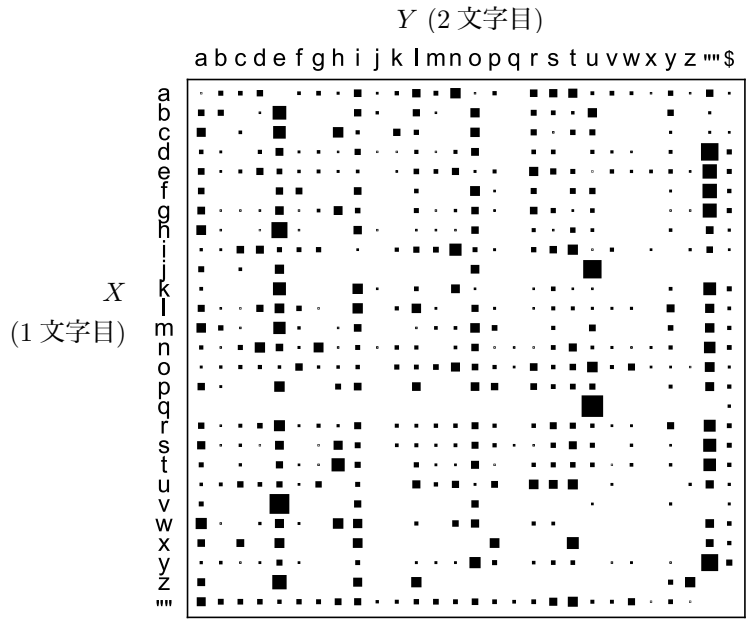


図 2.5: 『不思議の国のアリス』の文字バイグラム条件つき確率 $p(Y|X)$ を表す行列. 確率の総和は各行で 1 になっており, 興味深いパターンがみられることがわかります. 同時確率を表す図 2.3 と比べてみましょう.

```
if y in p[x]:
    c[x][y] = p[x][y] / p0[x] # 条件つき確率を計算
else:
    c[x][y] = 0
```

図 2.5 に, `alice.txt` からこうして計算した条件つき確率 $p(Y|X)$ をすべての文字 X, Y について示しました. 式(2.19)から, これは図 2.3 の $p(X,Y)$ を $p(X)$, すなわち各行の和で割ったものですから, $p(Y|X)$ は図 2.3 を行方向に正規化して和を 1 にしたものになっています.

2.4.1 確率の連鎖則

条件つき確率は公式 (2.19) で計算できますが, ただし, この公式を暗記する必要はありません. 条件つき確率 $p(Y|X)$ は, 「X が起きたときに Y が起きる

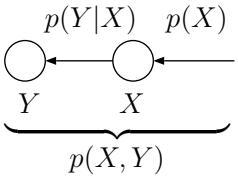


図 2.6: 確率の連鎖則 $p(X,Y)=p(Y|X)\,p(X)$ のイメージ. X と Y が両方起こるとは, まず X が起こり, その下でさらに Y が起きることと同じです.

確率」を表しているのでした. $p(X,Y)$ は「X と Y が同時に [*13] 起きる」確率のことですが, これは「まず X が起きてから, その条件の下でさらに Y が起きる」ことと同じです. つまり, 条件つき確率の意味から,

$$p(X,Y) = p(Y|X)\,p(X) \qquad (\textbf{確率の連鎖則}) \tag{2.20}$$

が明らかに成り立ちます. これを**確率の連鎖則** (chain rule) といいます. この様子を, 図 2.6 に示しました.

$p(X)\neq 0$ ならば, 式(2.20)の両辺を $p(X)$ で割れば公式式(2.19)が簡単に得られますから, 条件つき確率の公式(2.19)は自明な連鎖則(2.20)から簡単に得られます. よって, 公式 (2.19) を暗記する必要はなく, 慣れるまでは式(2.20)からその場で導くとよいでしょう. 条件つき確率の公式 (2.19) は確率の連鎖則 (2.20) から明らかですが, その直感的な意味は, 式(2.17)に示したように, 同時確率 $p(X,Y)$ を注目している条件 X について正規化して, 相対的な値を考えるということです.

なお,

$$p(X,Y) = p(X)p(Y) \qquad (\textbf{独立な事象の同時確率}) \tag{2.21}$$

が成り立つ場合, つまり式(2.20)と見比べれば

$$p(Y|X) = p(Y) \tag{2.22}$$

---

*13　慣例的に joint probability は「同時確率」または「結合確率」と訳されていますが, 「同時」というのは, 必ずしも時間的に同時刻に起きるという意味ではなく, 「両方起こる」という意味です.

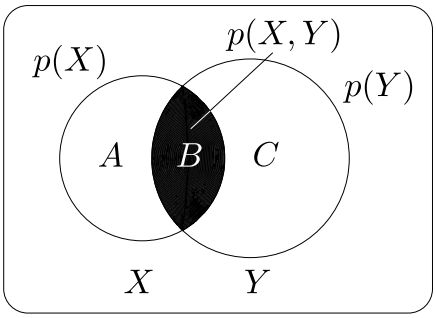


図 2.7: ベン図で表した確率と同時確率, および条件つき確率の関係. 枠で囲った全体の確率を 1 としたとき, 同時確率 $p(X,Y)$ は B の面積で, 条件つき確率 $p(Y|X)$ は割合 $B/(A+B)$ のことです. $p(X)$ を表す面積 $A+B$ に条件つき確率 $p(Y|X)$ をかければ B の面積, すなわち $p(X,Y)$ が得られます.

が成り立つ場合, X と Y は**独立**である, といいます. 式(2.22)は, X と Y が独立ならば X が与えられた下での Y の確率 $p(Y|X)$ はもともとの Y の確率 $p(Y)$ と等しい, つまり Y の確率は X の確率に影響を受けないことを表しているわけです. このとき, 式(2.21)のように X と Y の同時確率 $p(X,Y)$ は単にそれぞれの周辺確率 $p(X)$ と $p(Y)$ の積で表すことができます.

面積でみる確率 こうした条件つき確率と同時確率の関係は, 図 2.7 のようなベン図で表してみるとわかりやすいでしょう. 外側の四角で囲われた全体の面積を確率の総和である 1 とすると, 2 つの円の面積はそれぞれ, X と Y が起きる確率 $p(X)$ と $p(Y)$ を表しています. 黒で示した重なりが, X と Y の同時確率 $p(X,Y)$ です. 2 つの円で区切られた 3 つの領域を図のように A,B,C とおくと, $p(X)=A+B$, $p(Y)=B+C$, $p(X,Y)=B$ となっています.

このとき, X が起きた中で Y が起きる確率 $p(Y|X)$ は $\dfrac{B}{A+B}$ で, これは $\dfrac{p(X,Y)}{p(X)}$ を意味します. 逆に割合 $p(Y|X)=\dfrac{B}{A+B}$ が先にわかっていれば, 真ん中の領域の面積は X の面積にこれをかければよく, $(A+B)\cdot\dfrac{B}{A+B}=B$ となるわけです. これはつまり, $p(X)\cdot p(Y|X)=p(X,Y)$ を意味しています.

条件つき確率と頻度　なお, 実はわれわれの場合は, 条件つき確率はバイグラムの頻度から直接計算することができます. 表 2.1 および図 2.3 のもとになっているバイグラムの頻度は, $n(\mathtt{aa})=1$, $n(\mathtt{ab})=214$, $n(\mathtt{ac})=157$, $\cdots$, $n(\mathtt{a\$})=11$ で, その総和は $n(\mathtt{a})=8791$ です. つまりこれは, a が現れた 8791 回のうち, c が続いたのは 157 回だったということですから, $p(\mathtt{c}|\mathtt{a})$ は

$$p(\mathtt{c}|\mathtt{a}) = \frac{n(\mathtt{ac})}{n(\mathtt{a})} = \frac{157}{8791} = 0.0179 \tag{2.23}$$

と求めることができます.

これはなぜかというと, 頻度 $n()$ を使った式(2.23)は, 同時確率を使った式(2.19)と等価になるからです. 式(2.1)および式(2.4)から

$$p(\mathtt{ac}) = \frac{n(\mathtt{ac})}{N}, \quad p(\mathtt{a}) = \frac{n(\mathtt{a})}{N} \tag{2.24}$$

ですから, 式(2.23)は

$$\frac{n(\mathtt{ac})}{n(\mathtt{a})} = \frac{\dfrac{n(\mathtt{ac})}{N}}{\dfrac{n(\mathtt{a})}{N}} = \frac{p(\mathtt{ac})}{p(\mathtt{a})} = p(\mathtt{c}|\mathtt{a})$$

となり, 条件つき確率に等しくなります. このため, 以下では一般に, 文字列 h の後に文字 c が現れる条件つき確率は, 頻度 $n()$ を用いて

$$p(c|h) = \frac{n(hc)}{n(h)} \tag{2.25}$$

として計算することにします. 条件となる部分は後でみるように 1 文字とは限りませんので, 履歴 (history) という意味で, h で表しています.

2.4.2　ベイズの定理

条件つき確率を用いると, たとえば文字 j の後に文字 a が来る確率 $p(\mathtt{a}|\mathtt{j})$ は, 式(2.23)のように計算することができます. それでは逆に, j の前に来ることができる文字には, どんなものがあるでしょうか [*14].

*14　筆者は大学時代にエスペラント研究会に所属していましたが, 世界共通語を目指して作られた人工語であるエスペラント語では, j は複数を表す接尾辞で, libroj (本たち) や ĉiuj (すべて)

確率で書けば, これは, 前に来る文字を x として

$$p(\mathtt{j}|x) \qquad (x \text{ は前に来る任意の } 1 \text{ 文字}) \tag{2.26}$$

を求めたい, ということです. 式(2.26)はテキストで $x\mathtt{j}$ となる確率で, 順番が式とは逆になっていることに注意してください. 条件つき確率の定義式(2.19)から, 上の確率は

$$p(\mathtt{j}|x) = \frac{p(\mathtt{j}, x)}{p(x)} \tag{2.27}$$

と表すことができます. ここで分子に式(2.20)の確率の連鎖則を用いれば,

$$p(\mathtt{j}|x) = \frac{p(\mathtt{j}, x)}{p(x)} = \frac{p(x|\mathtt{j})p(\mathtt{j})}{p(x)} \tag{2.28}$$

と分解することができます. すべての文字 x について $p(x|\mathtt{j})$ は式(2.23) のように計算することができ, 文字のユニグラム確率 $p(\mathtt{j})$, $p(x)$ は 2.1 節で計算したように文字の頻度から簡単に求まりますから, 式(2.28)は簡単に計算することができます.

たとえば, 式(2.23)より $p(\mathtt{a}|\mathtt{j}) = 0.0410$ ですが, 式(2.1)より $p(\mathtt{j}) = 0.0011$, $p(\mathtt{i}) = 0.0662$ なので,

$$p(\mathtt{j}|\mathtt{a}) = \frac{p(\mathtt{a}|\mathtt{j})p(\mathtt{j})}{p(\mathtt{a})} = \frac{0.0410 \times 0.0011}{0.0662} = 0.00068 \tag{2.29}$$

と計算されます. 同様にしてほかの文字についても計算してみると,

```
r = {};
for x in chars: # c[x][y] = p(y|x)
    r[x] = {}   # r[x][y] = p(x|y)
    for y in chars:
        if y in c[x]:
            r[x][y] = c[x][y] * p0[x] / p0[y]
        else:
            r[x][y] = 0
r['j']
⇒ {'a': 0.0006825162097599817,
```

のように頻繁に使われています. しかし英語では, j が単語の末尾に用いられることは稀です. それでは, j はどんな場合に使われているのでしょうか?

```
'l': 0.0,
'i': 0.0,
'c': 0.00041701417848206843,
'e': 0.0014737307493920865,
's': 0.0,
 :
'u': 0.02943722943722944,
'r': 0.0,
'w': 0.0,
'o': 0.002087170042971148,
 :
'z': 0.0}
```

となり, 英語では, “uj” および “ej” の確率が比較的高いことがわかります.

式(2.29)をよく見ると, これは条件つき確率 $p(\mathtt{j}|\mathtt{a})$ を, 逆方向の条件つき確率 $p(\mathtt{a}|\mathtt{j})$ を使って計算することができる, ということを意味しています. こうして条件つき確率を引っくり返すことができる式(2.29), あるいはより一般に

$$p(X|Y) = \frac{p(Y|X)p(X)}{p(Y)} \qquad (\textbf{ベイズの定理}) \tag{2.30}$$

を, **ベイズの定理** (Bayes' Theorem) といいます. [*15] ベイズの定理を使えば, 条件つき確率 $p(X|Y)$ を逆の条件つき確率 $p(Y|X)$ で表すことができます. たとえば X が「原因」, Y が「結果」のとき, 結果 Y がわかったときの原因 X の確率 $p(X|Y)$ は, 原因から結果が得られる確率 $p(Y|X)$ を使って計算できる, ということになります.

このベイズの定理も, 本質的には確率の連鎖則の式(2.20)からすぐに得られますので, これを暗記する必要はありません. 慣れるまでは, $p(X,Y) = p(X|Y)p(Y)$ の両辺を $p(Y)$ $(\neq 0)$ で割って

$$p(X|Y) = \frac{p(X,Y)}{p(Y)} = \frac{p(Y|X)p(X)}{p(Y)}$$

---

*15　この公式の特別な場合は英国の牧師 Thomas Bayes (c.1701–1761) によって発見され, 死後に友人によって発表されため, ベイズの定理と呼ばれています. ただし, 一般化や実際の発展は, その後にフランスの数学者ラプラス (1749–1827) によって行われたものです[10]. とはいえ, ベイズの公式を積極的に利用するベイジアンにとって, ロンドンにあるベイズの墓石は今でも聖地となっています.

として, その場で導くといいでしょう.

ベイズの定理に慣れるため, もう少し練習してみましょう. インフルエンザ (以下インフル) が日本中で流行していた, ある年の冬, 普段は感染しない筆者も, 39 度近い高熱がまったく下がらない状況になりました. 急いで休日診療の医院にかかったところ, 何と検査結果はインフル陰性でした. ただし, 診察を担当したのは新人の医師で, 鼻の粘膜の入り口しか取らない簡易的なものでしたので, この結果にはかなり疑いが残りました. この高熱は本当に, ただの風邪なのでしょうか.

真の状態を表す変数を X, その時の診断を Y とおきましょう. 0 はインフル陰性, 1 はインフル陽性を意味します. 上記の症状や周囲の感染状況から, 自分の見立てでは, インフルである確率は 8 割はあると思っていました. 数式で書くと, $p(X=1)=0.8$, $p(X=0)=0.2$ ということになります [*16]. 一方で検査結果は陰性, つまり $Y=0$ だったわけです. 新人医師が誤診する確率は, 本当は追跡すれば客観的にわかりますが, ここでは図 2.8 のようだったとしましょう. つまり, 単なる風邪ならば, 医師の診断でインフルと誤診されることはないが (つまり $X=0$ のとき, $p(Y=0|X=0)=1, p(Y=1|X=0)=0$), 検査が甘いため, インフルの場合でも, 3 割程度は陰性と判断されてしまう (つまり $X=1$ のとき, $p(Y=0|X=1)=0.3, p(Y=1|X=1)=0.7$) と仮定します.

$X \setminus Y$	0	1
0	1	0
1	0.3	0.7

図 2.8: $p(Y|X)$ の表. X は真の状態を, Y は診断を表します.

さて, 実際の診断は上記の通り $Y=0$ (陰性) でしたので, 知りたいのはこのときの真の状態 X の確率, すなわち

$$p(X|Y=0) \tag{2.31}$$

です. 式 (2.19) の条件つき確率の定義から, この式 (2.31) は次のように書き直すことができます.

---

*16 ここでは説明のために事前確率を天下りに設定していますが, 本書でこの後扱うように, 実際に使う事前確率はできるだけ無情報にしたり, それ自体をデータから学習するなど, 客観的に決められるものです. この場合も, 同様の症状だった人の診察記録を多数集めれば, 診断以前の事前確率 $p(X)$ を計算することができるでしょう. よって, ベイズ統計が「主観的な事前確率を使う統計」であるとする批判は, 実際にはあまり正しくありません.

(2.32)
$$p(X|Y=0) = \frac{p(X,Y=0)}{p(Y=0)} = \frac{p(X,Y=0)}{\sum_X p(X,Y=0)} = \frac{p(Y=0|X)p(X)}{\sum_X p(Y=0|X)p(X)}$$

X は具体的には 0 か 1 ですから, 式(2.32)に上の値を代入すると,

(2.33)
$$\begin{cases} p(X=0|Y=0) &= \dfrac{p(Y=0|X=0)p(X=0)}{p(Y=0|X=0)p(X=0)+p(Y=0|X=1)p(X=1)} \\ &= \dfrac{1\times 0.2}{1\times 0.2+0.3\times 0.8} = \dfrac{0.2}{0.2+0.24} = \mathbf{0.455} \\ p(X=1|Y=0) &= \dfrac{p(Y=0|X=1)p(X=1)}{p(Y=0|X=0)p(X=0)+p(Y=0|X=1)p(X=1)} \\ &= \dfrac{0.3\times 0.8}{1\times 0.2+0.3\times 0.8} = \dfrac{0.24}{0.2+0.24} = \mathbf{0.545} \end{cases}$$

となります. すなわち, 検査は陰性 ($Y=0$) でしたが, 本当はインフルである ($X=1$) 確率は半分以上の 55%もある, ということがわかります. 実際この数日後, あまりに熱が下がらないので別のベテランの先生の医院で再検査したところ, 明らかにインフルということがわかり, タミフルを処方されてすぐに熱は収まりました.

式(2.33)の計算をよく見ると, どちらも分母は同じで $(A+B)$ の形をしており, 確率はそれぞれ $A/(A+B)$, $B/(A+B)$ の形をしています. すなわち, 式(2.33)を求めるためには分子である A と B, つまり $p(X,Y=0)=p(Y=0|X)p(X)$ を $X=0$ と 1 の場合について求めればよく, それを和が 1 になるように正規化すれば確率が得られる, ということがわかります.

これは, ベイズの定理を使う場合すべてについて成り立ちます. なぜならば, 条件つき確率の定義から

(2.34)
$$p(X|Y) = \frac{p(X,Y)}{p(Y)}$$

ですが, ここで分母の $p(Y)$ は X には関係しない, 和を 1 にするための定数だからです. よって, ベイズの定理は比例を表す記号 $\propto$ を使って,

$$p(X|Y) \propto p(X,Y) \qquad \text{(ベイズの定理 (同時確率版))} \tag{2.35}$$

とも表すことができます. または, 右辺を確率の連鎖則 $p(X,Y)=p(Y|X)p(X)$ で分解すれば,

$$\underbrace{p(X|Y)}_{\text{事後確率}} \propto \underbrace{p(Y|X)}_{\text{尤度}} \underbrace{p(X)}_{\text{事前確率}} \qquad \text{(ベイズの定理 (比例版))} \tag{2.36}$$

と書き直すことができます. 実際の計算は, こちらで行えばよいでしょう. たとえば式(2.33)の計算は, 実際には

(2.37)

$$\begin{aligned} p(X|Y=0) &\propto p(Y=0|X)p(X) \\ &= \begin{cases} p(Y=0|X=0)p(X=0) \\ p(Y=0|X=1)p(X=1) \end{cases} = \begin{cases} 1 \times 0.2 \\ 0.3 \times 0.8 \end{cases} = \begin{cases} 0.2 \\ 0.24 \end{cases} \end{aligned}$$

となり, これを和が 1 になるように正規化して

$$\begin{cases} p(X=0|Y=0) &= \dfrac{0.2}{0.2+0.24} = 0.455 \\ p(X=1|Y=0) &= \dfrac{0.24}{0.2+0.24} = 0.545 \end{cases} \tag{2.38}$$

と, 簡単に求めることができます.

式(2.36)は文字通り, Y が与えられたときの X の確率は, X だけの確率 $p(X)$ に, X から Y が観測される確率 $p(Y|X)$ をかけたものに比例することを表しています. Y が与えられた後の確率なので, 左辺の $p(X|Y)$ を**事後確率** (posterior probability), これに対して右辺の $p(X)$ を**事前確率** (prior probability) ともいいます. $p(Y|X)$ は $X \to Y$ となる確率ですが, いま Y は与えられているため, これは X の関数とみることができ, X の**尤度関数** (likelihood function) といいます. 尤度(ゆうど)とは, 観測値 Y について X がどんな値をとれば尤(もっと)もらしいのか, を

表しています. よってベイズの定理は, 言葉で書けば

(2.39) (事後確率) $\propto$ (尤度) $\times$ (事前確率)

の形で書くことができます. 式(2.35)や式(2.36)の変形はこの後で頻繁に使いますので, 覚えておいてください. ベイズの定理にさらに慣れるため, 本章の演習問題を解いてみるとよいでしょう (→演習 (10)).

2.5 文字nグラムモデル

2.5.1 文字列の確率的生成

こうして文字の条件つき確率がわかると, 文字列を確率的に生成することができます. そのためにまず, 文字列の確率について考えてみましょう.

たとえば文字列 s が cat のように 3 文字だったとき, 1 文字目, 2 文字目, 3 文字目を表す確率変数をそれぞれ X, Y, Z とおくと, s の確率は

$$p(s) = p(X, Y, Z) \tag{2.40}$$

です. たとえば $p(\mathtt{cat}) = p(X = \mathtt{c}, Y = \mathtt{a}, Z = \mathtt{t})$ になります. このとき確率の連鎖則式(2.20)から, まず X, Y をまとめて 1 つにして考えれば, 式(2.40)は

$$\begin{aligned} p(s) &= p(X, Y, Z) \\ &= p(Z|X, Y)p(X, Y) \end{aligned} \tag{2.41}$$

と分解することができます. ここでさらに $p(X, Y)$ にも確率の連鎖則を用いれば,

$$\begin{aligned} p(s) &= p(X, Y, Z) \\ &= p(Z|X, Y)p(X, Y) \\ &= \underbrace{p(Z|X, Y)}_{\text{(A)}} \underbrace{p(Y|X)}_{\text{(B)}} p(X) \end{aligned} \tag{2.42}$$

と文字列の確率を積に分解することができます. この確率をどう計算するかは, 何グラムのモデルを考えるかによります.

ユニグラムの場合

ユニグラムを考える場合は, 文字の確率はそれより前に出現した文字に依存しませんから, 式(2.42)の最後の行の (A)(B) はそれぞれ

$$\text{(A)} \quad p(Z|X,Y) = p(Z)$$
$$\text{(B)} \quad p(Y|X) = p(Y)$$

となります. よって

$$p(s) = p(X,Y,Z) = p(X)p(Y)p(Z) \tag{2.43}$$

です. つまり, ユニグラムで文字列 s を生成するには, 1 文字目, 2 文字目, 3 文字目をそれぞれ単にユニグラム分布から生成すればよいわけです.

2.1 節のように Python の辞書 `p` に文字 `c` の確率が `p[c]` として保存されているとき, この中から確率に従ってランダムに 1 文字を選ぶ関数 `genchar` は, $0 \leq r < 1$ の一様乱数を生成する関数 `rand()` を使って,

```
from numpy.random import rand
def genchar (p):
    s = 0
    r = rand()
    for c in p: # 文字 c を順番に調べる
        s += p[c]
        if (r < s):
            return c
    return c    # ここには来ないはずですが，念のため
```

のように書くことができます. [*17] よって, 長さ N の文字列 `s` を生成するには,

```
s = ""
for n in range(N):
    s += genchar(p)
```

とすれば行うことができます.

こうして『不思議の国のアリス』のユニグラムから生成した長さ 40 の文字列は, 下のようになりました. ここから後は, 句読点や記号も含んだ元のテキスト `alice.full.txt`[*18] を使って確率を計算しています.

---

*17 詳しい方への注：この方法の計算量は, カテゴリ数 K について $O(K)$ です. もしサンプルする確率分布が固定されている場合には, Alias 法とよばれる方法で事前にテーブルを作っておけば, 以後は $O(1)$ でサンプルすることができます[11]. さらに 2020 年になって, Fast Loaded Dice Roller (FLDR) とよばれる手法が提案され[12], 情報理論的限界に近い速度でサンプリングすることが可能になりました. C と Python による FLDR の実装は, `https://github.com/probcomp/fast-loaded-dice-roller` で公開されています.

*18 サポートサイトの同じフォルダに置いてあります.

```
    % unigram.py alice.full.txt alice.1gram
    % unigram-gen.py alice.1gram 5 40
⇒ !reyrnh  r'l ls ses aso oeheh   s ‘nreai
   rnnhntmls  ga tfi ht-ltreat wag ‘areia
   lhne infeeulctrmwohno u ! oe ao n iwssvd
   fv rath tfegyanc!ytomst.d tcoa ni,us,oe'
   t,eeehiiapscoitna -gpeksag,,tnlsoemw si‘
```

上記はサポートサイトの unigram.py で文字のユニグラム確率を計算してファイル alice.1gram に保存した後, unigram-gen.py で生成しています. 文字を完全に独立にとらえていますので, およそ英語のようには見えませんが, e や t といった文字の頻度が高くなっており, 文字ユニグラム確率を反映しているようです. これは, 完全にすべての文字を等確率に選ぶ場合 (**0 グラム**といわれます) と比べればはっきりするでしょう. 0 グラムから生成すると, 同じ場所にある zerogram.py を使って以下のようになります.

```
    % zerogram.py alice.full.txt 5 40
⇒ ul!'_jx'‘fi!k_(x_l‘whs.;(u'_ bm'ucj!‘b?.
   .cxba !r'ciikj[z;:obu;‘m',ig!cngle !n''o‘
   ?]n?g.nrpw)kw(fucpxgouj,fv.ibyvuf-rbx;m_
   ‘‘]:?gth:z(gthhx]o-qq-'lh!f[jo''mujo!si[.!
   u(its,y,_hqiqnaxbic. i qepzy[-nt.''m(sm,‘
```

こうして, 一見ランダムに見えるユニグラムも, 0 グラムと比べればずっと自然言語の統計を反映している, ということがわかりました.

バイグラムの場合

バイグラムの場合は連続した 2 文字を考慮しますので, 式(2.42)式の (A)(B) の確率は, 以下のようになります.

$$\begin{aligned} &\text{(A)} \qquad p(Z|X,Y) = p(Z|Y) \\ &\text{(B)} \qquad p(Y|X) = p(Y|X) \end{aligned}$$

つまり, 3 番目の文字 Z は Y とのバイグラム YZ にのみ関係しており, X には依存しないことになります. よって s の確率は,

$$p(s) = p(X,Y,Z) = p(Z|Y)p(Y|X)p(X) \tag{2.44}$$

で与えられます. したがって, 式(2.44)式から s を生成するには,

(1) まずユニグラム $p(X)$ から 1 文字目 X を生成する
(2) 生成された X を使って, $p(Y|X)$ から 2 文字目 Y を生成する
(3) 生成された Y を使って, $p(Z|Y)$ から 3 文字目 Z を生成する

とすればいいことになります.

これでもよいのですが, ただし, 上のステップ (1) には注意が必要です. 多くの教科書にはマルコフモデル (ここではバイグラム) から生成するには上記のようにすると書かれていますが, このままナイーブに行うと, ユニグラム確率すなわち,「文字の一般的な確率」$p(X)$ から最初の文字 X を生成することになってしまいます. たとえば, 日本語ですべての文字列が引用符 “「” から始まっていたとしても, 確率の高い “の” や “が” が最初の文字として生成される可能性が高くなってしまうわけです. これは明らかに不合理ですね.

そこで自然言語処理では一般に, 文字列や単語列の最初と最後に, 文頭と文末を表す見えない特殊な文字 (あるいは単語) “^” と “$” があると考えます. ここで “^” や “$” は説明のための便宜的な表記で, 実際にはテキストに現れない文字 (たとえば “\x1c”) や単語 (空文字列 “” や “_BOS_”) を用います. *19 実装例は, サポートサイトの bigram.py や trigram.py を参照してください. これらは, BOS (beginning of sentence) および EOS (end of sentence) といわれることもあります. *20

よってこの場合, 文字列 $s =$ ”alice” は

```
"^alice$"
```

として扱います. このとき文頭文字 ^ はすでに与えられているとしますので, 式(2.44)式の $p(X)$ は 1 です. したがって, s の確率は

$$p(s) = p(\mathtt{a}|\mathtt{\hat{}})\, p(\mathtt{l}|\mathtt{a})\, p(\mathtt{i}|\mathtt{l})\, p(\mathtt{c}|\mathtt{i})\, p(\mathtt{e}|\mathtt{c})\, p(\$|\mathtt{e}) \tag{2.45}$$

と, すべてバイグラムで書けることになります. 最後に文末文字 “$” を導入して文末との接続確率 $p(\$|\mathtt{e})$ を考えているため,「文字列の最後が “e” で終わり

*19 文字 “\x1c” は ASCII コード表の FS (field separator) ですが, 他の文字でもかまいません. なお, 16 進数で\x1c=8 進数で\034 の文字 (Ctrl-\) は, テキスト処理言語である awk や perl では, 擬似的に多次元配列を実現するための標準のセパレータとして内部で使われています.

*20 実は理論的には, BOS と EOS は同じでも問題ありません. 文字や単語は BOS からは出るだけ, EOS へは入るだけだからです. 高速な実装で文字や単語を整数の ID で管理している場合は, EOS には 0 や -1 (二進数の補数表現で 1111…1) を用いるとよいでしょう.

やすい」といった言語的性質も考慮できるのが特徴です. この点は, 4.4 節で隠れマルコフモデル (HMM) を扱う際にも再び議論します.

上記のようにして alice.txt の文字バイグラムから生成した文字列を次に示しました (bigram.py, bigram-gen.py). この場合, 文末文字 "$" が生成されればそこで文字列が終わるという合図ですので, 生成する長さを指定する必要はありません.

```
% bigram.py alice.full.txt alice.2gram
% bigram-gen.py alice.2gram 5
⇒ no as pe ililles arugl ster, cake erue!
‘stick, ce w illite woprud p, jenthing bs washiofopor ld ur
wheshig,’sey bes elor.
[l that (‘y the tcepane uth wean to, nthiskerus amerowhexi-
gokem serghipp as bo wsime’thertord alork athin t itwas wh
se f en.
wrsal d owhoughen trnndsprownore an bsoupandolsingry, ucane,
’ bioull th be wide s ind oxifunde.
```

0 グラムやユニグラムと比べると, だいぶ英語に近くなってきました. ただ, バイグラムは「隣り同志の文字の確率」しか考えていないので, もう少し賢くする必要がありそうです.

メモ：文字列の確率と EOS

文字列の終わりを表す特殊文字 EOS を考えることは, 文字列がどんな風に終わりやすいかという言語の性質を表せるだけでなく, 実は, 確率モデルとしても不可欠な要素です. 図 2.9 に示したように, EOS を考えると, “”(空文字列), “a”, “aa”, “ab”, … といったすべての文字列は先頭から順に文字を選択し, EOS が出たときそこで止まった結果として表すことができます. あらゆる文字列はこうした選択の結果に対応していますから, それらの確率の総和は必ず 1 になります. いっぽう, EOS がないと “aaaaa…” といった, いくらでも長い文字列にも一定の確率を割り当てることができ, それらが無限にありますから, 確率の総和は容易に 1 を超えてしまいます. これは, 文字列の確率モデルとしては不適切です. *21

なお, 再帰的な選択によって文字列を生成できることから, 次の文字の選択肢を可視化して, その幅を確率に比例させれば, $[0,1)$ の範囲の実数を次々と指示することで, 効率的に文字を入力することができます *22. 図 2.10 に示したケンブリッジ大学の MacKay らによる Dasher [13] *23 は, このことを利用した入力システムで, もし病気や障害で指が動かない場合でも, まぶたの上下や, 呼吸による腹の上下といった 1 次元のわずかな信号さえあれば, 言葉を発することができます.

これらは文字列に限らず, 3 章の単語列の場合でもすべて同様に成り立ちます. *24

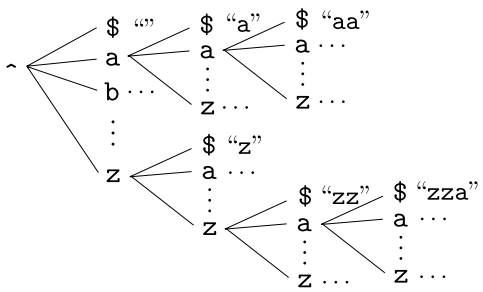


図 2.9: 文字列全体の空間を表す樹形図. ^ で BOS, $ で EOS を表しています.

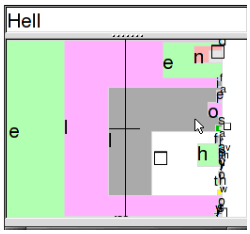


図 2.10: 入力システム Dasher で “Hello” を入力している様子. 1 次元の選択を次々に行うだけで, 文字列を入力することができます.

*21 深層学習による現代の言語モデルがこの条件を満たしているかについては, 無限長の文字列

2.5.2　ゼロ頻度問題

バイグラムでは “er” や “al” のような隣りあう 2 文字の連続を考えてきましたが, もっとよい言語のモデルとするには, “ing” や “has” のように 3 文字の連続を考えればよさそうです. こうした 3 文字の連続を, 3 グラムまたは**トライグラム** (trigram) といいます. 一般に, 0 グラム, 1 グラム, 2 グラム, 3 グラム, …をまとめてn**グラム**とよび, nグラムの条件つき確率に基づいて文を生成する確率モデルを, n**グラム言語モデル**といいます. nグラム言語モデルは, n 文字の連続をモデル化するものになっています.

文字トライグラム言語モデルでは, バイグラムのように直前の 1 文字を見て

$$p(Y=\mathtt{i}|X=\mathtt{e}) \tag{2.46}$$

のように条件つき確率を計算する代わりに,

$$p(Z=\mathtt{i}|X=\mathtt{a},Y=\mathtt{l}) \tag{2.47}$$

のように, 直前の 2 文字を見て条件つき確率を計算します.[*24] 文頭を表す特殊文字 `"^"` は, この場合, 最初の文字のために 2 個必要になり, 文字列 `"alice"` は `"^^alice$"` と思って a から確率を計算します. 式(2.25)と同様に頻度 $n()$ を用いて表せば, トライグラム確率は

$$p(z|x,y)=\frac{n(xyz)}{n(xy)} \tag{2.48}$$

と書くことができます. ここで x は 2 つ前, y は 1 つ前の文字で, z は予測する文字です.

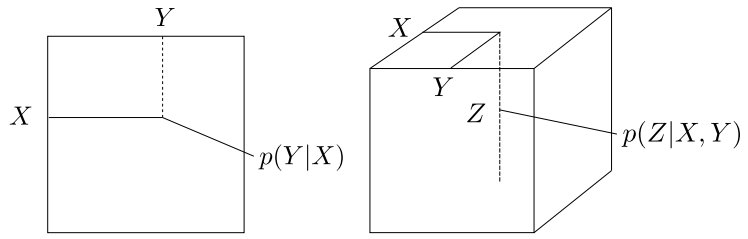
ただし, この場合はユニグラムやバイグラムと異なり, 大きな問題が発生します. 式(2.46)のバイグラムの確率は X と Y の組み合わせの数, つまり文字種の

に対する確率を考える必要があることから, 測度論を用いた最近の研究[14]によって解析されています.

*22　これは情報理論では, **算術符号**とよばれる符号化を行っていることに相当しています.

*23　`http://www.inference.org.uk/dasher/` に日本語版を含む実装や情報があります.

*24　同様に 4 グラムや 5 グラムも考えることができますが, 取り扱いが複雑になるため, 詳しくは 3 章を参照してください. 筆者はノンパラメトリックベイズ法に基づき, 文脈によって異なる n を学習することで, n の指定を不要にした ∞ グラム言語モデルを提案しています[15][16].



(a) バイグラムの場合: V^2 個　　(b) トライグラムの場合: V^3 個

図 2.11: nグラムの予測確率の組み合わせ. 実線は条件となる変数を, 破線は予測する変数を表しています. V は語彙の数 (文字nグラムなら文字の種類数) です.

2 乗だけ存在します. 英語の場合は文字種は記号を入れても最大でも 256 種類程度ですから [*25], バイグラムでは図 2.11(a) のように, 最大 $256^2 = 65,536$ 個の確率がテキストの文字の頻度から計算できればよいわけです.

しかし, トライグラムの場合は図 2.11(b) のように, 式(2.47)のトライグラムの確率は X, Y, Z の組み合わせで, 最大で $256^3 = 16,777,216$ 個存在します. 『不思議の国のアリス』は 132,656 文字でしたから, これは可能な組み合わせの 0.8%にすぎず, もし図 2.11(b) のセルに頻度を 1 ずつ均一に置いたとしても, すべての組み合わせに頻度を埋めることはできません. 実際には頻度は `e` や `a` などに偏っていますから, この傾向はさらに極端で, 図 2.11(b) ではほとんどのセルの頻度は 0 だということになります. これを, **ゼロ頻度問題**といいます.

ゼロ頻度問題のため, トライグラム言語モデルはそのままでは大きな問題が生じます. たとえばトライグラム `"onk"` は `alice.txt` には一度も現れませんので, 式(2.48)より

$$p(\mathtt{k}|\mathtt{on}) = \frac{n(\mathtt{onk})}{n(\mathtt{on})} = \frac{0}{n(\mathtt{on})} = 0 \tag{2.49}$$

です. そうすると, たとえば文字列 `"monk"` の確率は

*25 ASCII 文字は 8 ビット目が 0 なので 128 種類で収まりますが, アクセント記号が存在するヨーロッパ系の言語 (Latin-1 文字セット) では 8 ビット目が 1 の場合があるため, 最大は 256 文字になります.

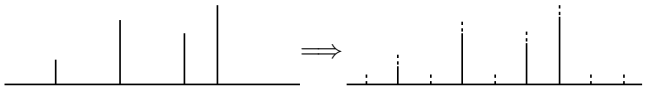


図 2.12: 確率分布の平滑化. 頻度にすべて小さな値 α を足して正規化することで, 離散的な確率分布がより「滑らか」になっています.

$$p(\texttt{"monk"}) = p(\texttt{m}|\texttt{\^{}\^{}}) \cdot p(\texttt{o}|\texttt{\^{}m}) \cdot p(\texttt{n}|\texttt{mo}) \cdot \underbrace{p(\texttt{k}|\texttt{on})}_{=0} \cdot p(\$|\texttt{nk}) = 0 \tag{2.50}$$

になってしまい, "monk" や "monkey" の確率はすべて 0, つまり英語には絶対に現れない文字列ということになってしまいます. これは明らかにおかしいですね.

加算平滑化

そこで, 最も簡単な解決法として, トライグラムのすべての頻度に小さな値 α を加えることを考えてみましょう. これにより図 2.12 に示したように, 確率分布はより「滑らか」になるため, こうした操作を確率分布の**平滑化** (**スムージング**, smoothing) といいます.

こうすると, トライグラムの予測確率は式(2.25)に代えて, 次の式で求められます.

$$p(z|x,y) = \frac{n(xyz)+\alpha}{\sum_c (n(xyz)+\alpha)} = \frac{n(xyz)+\alpha}{n(xy)+V\alpha} \tag{2.51}$$

ここで V は文字種の数で, 可能な最大値は英語なら 256, 日本語なら 8500 程度ですが, 実際には学習するテキストに現れた異なり数を用いればよいでしょう [*26].

平滑化について詳しくは 3 章の単語 n グラムモデルで議論しますが, 単純に頻度に小さな値を加えるこの方法を, **加算平滑化** (additive smoothing) といいます. 加算平滑化は最も簡単な平滑化法ですが, トライグラムに直接用いる場合は, データ量によりますが, α はかなり小さな値にする必要があります.[*27] ただし,

*26　3 章の単語の n グラムモデルの場合, 可能な単語の数は無限にありますから, ナイーブに考えると V を無限大に取らなければならなくなってしまいます. よって, 学習データに出現した語彙の総数を用いるのが, 現実的な方法といえます. 筆者による NPYLM [17]では, 階層ベイズモデルを用いることで, 理論的に無限個の語彙を扱うことができます.

*27　トライグラムの場合は, ゼロ頻度問題から, 式(2.51)でほとんどの文字 z について $n(xyz)=$

$\alpha=0$ にすると平滑化を行わないため, ゼロ頻度問題が生じてしまいます. 一方で α を大きくすると, 式(2.51)の $p(z|x,y)$ は, z が何であっても一様分布 $1/V$ に近づいてしまいます. 以下の例では, 英語では $\alpha=$`1e-4`$=0.0001$, 日本語では $\alpha=$`1e-5`$=0.00001$ としました. このとき, 式(2.49)式の $p(\mathtt{k}|\mathtt{on})$ は式(2.51)より, `alice.full.txt` では $n(\mathtt{on})=1054$, 現れる文字の総数 $V=43$ なので

$$p(\mathtt{k}|\mathtt{on}) = \frac{0+0.0001}{1054+43\times 0.0001} = 0.00000009 \tag{2.52}$$

になり, 確かに 0 でない確率が割り当てられていることがわかります.

文字トライグラムからの生成

`alice.txt` に対して文字トライグラム言語モデルをサポートサイトの `trigram.py` で計算して保存し, `trigram-gen.py` で生成すると, 下のようになりました. バイグラムに比べて, 大幅によい言語モデルとなっていることがわかります. [*28]

```
% trigram.py alice.full.txt alice.3gram 1e-4
% trigram-gen.py alice.3gram 5
⇒ ‘who an ard to my frot an thisher, awlind the as ey pled the
low fousee me a onereepleake died of yound of sly and ance,
beithe pearniene was no paid they hargense, be fould thereep
on’ting i cat shrough al bes mocked es.’
‘of the anagaid rattlenly crioll, chink top on.
so the hat whe in al’s ithe could but of musibleake onse, a
that ‘‘youg, theress; ‘it bithe withe queence lice, yought
abloo ding tion, all mock thatinsed wou juse words tur a
plice fing suce rasheadveraclar!
```

---

0 です. z の確率を予測する際, これが V 通りのほとんどを占めますから, 頻度 0 の文字についての確率の総和はほぼ $V\cdot\dfrac{0+\alpha}{n(xy)+V\alpha} = \dfrac{V\alpha}{n(xy)+V\alpha}$ になります. これを書き直すと $1\Big/\left(\dfrac{n(xy)}{V\alpha}+1\right)$ になり, $V=10000$, $n(xy)=10$ のとき, $\alpha=0.01$ としても頻度が 0 の文字の確率の総和は $1\Big/\left(\dfrac{10}{10000\cdot 0.01}+1\right) = 1\Big/\left(\dfrac{10}{100}+1\right) \simeq 0.91$ となり, 頻度 0 の文字にほとんどの確率が割り当てられてしまうことになってしまいます.

*28 このため, 音声認識の分野では 1980 年代から長い間, 単語トライグラム (3 章) が標準的な言語モデルとして使われていました. 現在は, 必要な場合は LSTM や Transformer などのより高度な深層学習による言語モデルで文の確率を計算します.

夾竹桃の方に向ってしまった。またどたどたどたど切トっ、ランペシヤの中心地とて遠い所も見せて来る蓄音機の浪花節。わびしげしげと眺めたきりして来ていた。慌てている様子を噛む習慣もパラオ本島オギワルイという島の方が一面に糜爛した良いので、筋は良く知って来た。
海岸から二度と、うす汚い、この言うは何をあけた時から真白い裏を見るとは作るが。それら南方へ歩いた渚に踏出した翡翠色に向って、白い雨が頻りに来たあとではない。こんなと思う。人も住まないが、素晴らしい。

図 2.13: 中島敦『環礁』の文字トライグラムからの確率的生成結果.

「標本室にありましたら、またまえ。だけ青く茂ったり暗くなりましたけれどもはっきりに照らしい楽器の灯を、窓を見ていますけれども滑って、ジョバンニたちでいるものをひろっ務のから僕決して戻ろう。大きなとうに平らな。」カムパネルラと二熟、、カムパネルラを見ていました。
ジョバンニが、砂に三つの街燈の方へ急ぐのです。ジョバンニが左手の崖が川の水をわたしばらく、飛び出しても、顔を出しました。
銀河のお宮のけよってでした。二人も胸いって微かにうつくしくて、
ちょう。ぼくが、続いても押し葉にすきとお会いました。
「ほんといいとジョバンニのときすぎうっと天の川の水にあんな水は、すっかりにして校庭の隅の桜の木のあかりを川へ帰らずの鳥捕りがとまりました。

図 2.14: 宮沢賢治『銀河鉄道の夜』の文字トライグラムからの確率的生成結果. 文字の言語モデルなので, 「単語」という概念はありません.

現在は文字はユニコードによって言語を問わず扱えるようになっていますので, `trigram.py`, `trigram-gen.py` は英語以外のテキストに対しても適用することができます. [*29] 図 2.13 に, 中島敦『環礁』(229 行, 36419 文字) から文字トライグラムで生成した文を, 図 2.14 に宮沢賢治『銀河鉄道の夜』(459 行, 38292 文字) から同様に生成した文を示しました.

『銀河鉄道の夜』は若干ひらがなが多く, 文字トライグラムではモデル化し切れていない部分がありますが, 漢籍を背景にした中島敦の文体は長距離の依存性が少なく, 文体の特徴を比較的よくとらえていることがわかります.

単語のランダム生成

また, 単語は文字からなっていますから, `"alice"`, `"fortunately"` などの単

*29　日本語や中国語などでは「単語」をどう定義するかという問題がありますが, 文字の言語モデルは単語の定義に関係なく使うことができます.

語をそれぞれ「文」とみなせば, alice.txt にある 2748 語の語彙の文字列から, 新しい「単語」を文字nグラム言語モデルを使ってランダムに生成することができます. 下に, その様子を示しました. ここでは, 1 行に語彙の単語が 1 つずつ並んだファイル [*30] を alice.lex としています.

```
    % trigram.py alice.lex alice-lex.3gram 1e-4
    % trigram-gen.py alice-lex.3gram 20
⇒ que
    twer
    hadder
    stralkjuse
    late
    shatted
    vuld
    cut
    comished
    scretch
    flobs
    gar
    sely
    persed
    spoisoleds
    besersty
    ding
    arighen
    nage
    up
```

文字トライグラムを使っただけで, かなり英語らしい (が実際には存在しない) 単語が生成できていることがわかります.

*30 これは色々な方法で作成できますが, 最も簡単には, Mac や Linux, WSL には標準で含まれているテキスト処理言語 awk を使って, % awk '{for(i=1;i<=NF;i++)freq[$i]++}; END{for(w in freq) print w}' alice.txt > alice.lex とすると作ることができます.

メモ：nグラムモデルとマルコフモデル

39 ページで導入した n グラムモデルは, 文字の発生確率が直前の $(n-1)$ 文字に依存する確率モデルです. 一般に, 事象の発生確率が直前の事象だけに依存するとき, 確率論や情報理論では**マルコフ性**があるといい, **マルコフモデル**とよばれて議論されます. 特に, 直前の n 個の事象に依存する場合を, n 次のマルコフモデルといいます. すなわち, n グラムモデルとは$(n-1)$次のマルコフモデルのことです. 自然言語処理では, 2.2 節で `alice.txt` から文字の 2 グラムを取り出したように, n 個の文字の具体的な繋がりに興味があることも多いため, n グラムモデルという言い方をするのが普通です. マルコフモデルの名前は, ロシアの数学者 Andrej Markov (1856–1922) が, 1913 年の論文[18]でまさに言語の問題, すなわちプーシキンの小説『オネーギン』の最初の 20,000 文字を数えて, 母音と子音が連続する確率を計算したことによります [*31]. 第一次世界大戦より前のこの時代, まだコンピュータは登場していないことに注意してください.

2.6　統計モデルの学習と評価

2.6.1　学習データとテストデータ

前の 2.5.2 節で文字 n グラムの平滑化に用いた $\alpha=0.0001$ (英語), $\alpha=0.00001$ (日本語) は, 実は筆者が出力の様子を見て, 人手で決めた [*32] パラメータでした. $\alpha=0$ とすると, 2.5.1 節でみたようにデータに偶然出現しなかった n グラムの確率がすべて 0 になってしまい, 図 2.13 や図 2.14 といったテキストの確率は, 1 個でもそうした n グラムが含まれていれば 0 になってしまいます. 逆に $\alpha \to \infty$ と大きくすると, 式(2.51)から, n グラムの確率はすべて $1/V$ の一様な確率になってしまい, やはりテキスト全体の確率は低くなってしまいます. よって, α を変えれば図 2.15 のように, どこかにテキストの確率を最大化する $\alpha = \alpha^*$

*31　この研究は 20000 文字の統計を単に数えるのではなく, 100 文字の連続× 200 個に分け, それらの間での統計量のばらつきを計算しており, 現代のわれわれにも大変参考になるものです.

*32　もし人手で決める場合も, 41 ページの脚注に示したように, 数学的にある程度根拠のある値とすることは重要です.

があると考えられます. それでは, 最適な α^* を客観的に決めるにはどうすればよいのでしょうか.

いま, 我々の手元には `alice.txt` のようなテキストがありますから, α^* を決めるには, このうち一部のテキストを検証用に残しておき, 残りのテキストで n グラムモデルを学習して, 残しておいた検証テキストの確率が最大になるような α を求めればよいでしょう.

たとえば最も簡単な場合として, 文字ユニグラムモデルで, テキストが短い現代俳句 [*33]

ひいらぎをかをらせていつまでもいま (野口る理)

だったとしましょう. この文字列をランダムにシャッフルして

$$\underbrace{\text{いかをらせもでいひつまら}}_{D}\ \underbrace{\text{ぎいをまて}}_{D'}$$

とし, 前半の 12 文字 D から文字の確率を計算すると, 文字の頻度は $n(\text{ぎ})=0$, $n(\text{い})=2$, $n(\text{を})=1$, $n(\text{ま})=1$, $n(\text{て})=0$ ですから, 式(2.51)でユニグラムの場合を考えれば, ひらがなの総数は約 87 個 ($V=87$) ですから [*34],

$$\begin{cases} p(\text{ぎ}) = p(\text{て}) = \dfrac{0+\alpha}{12+87\alpha}\,, \\ p(\text{を}) = p(\text{ま}) = \dfrac{1+\alpha}{12+87\alpha}\,, \\ p(\text{い}) = \dfrac{2+\alpha}{12+87\alpha} \end{cases} \tag{2.53}$$

となります. よって後半の文字列 D' の確率は,

$$p(D'|D,\alpha) = p(\text{ぎ})\,p(\text{い})\,p(\text{を})\,p(\text{ま})\,p(\text{て}) \tag{2.54}$$

*33 元の句では「柊」だけが漢字ですが, ここでは説明のため平仮名にしています. 余談ですが,「柊をかをらせていつまでもいま」のこの句は,「いつまでも」の永遠性と「いま」の刹那性が「い」という同じ音を通じて意味的に共鳴し (Jakobson の詩学 [19]), それが「かをらせて」の微かに古典的な香りを通じて「柊」に象徴的に表されている, 素晴らしい現代俳句の一つだと思います.

*34 Unicode の Hiragana ブロック `https://unicode.org/charts/PDF/U3040.pdf` の表によります.

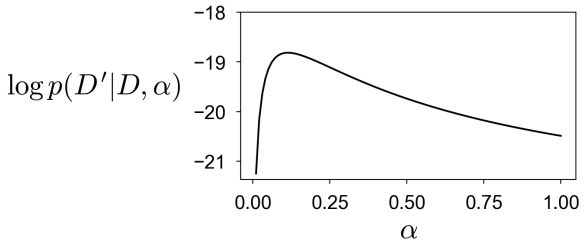


図 2.15: 俳句の仮想例での評価データ D' の予測確率の対数 $\log p(D'|D,\alpha)$ とパラメータ α. この場合, $\alpha=0.114$ で D' の確率が最大になっています.

$$= \left(\frac{\alpha}{12+87\alpha}\right)^2 \times \left(\frac{1+\alpha}{12+87\alpha}\right)^2 \times \left(\frac{2+\alpha}{12+87\alpha}\right)$$
$$= \frac{\alpha^2(\alpha+1)^2(\alpha+2)}{(12+87\alpha)^5}$$

になります. この確率の対数を, α についてプロットした図を図 2.15 に示しました.

式(2.54)は α の関数ですから, 勾配法を使ったり, 微分して 0 とおくことで極大値を求めることができます. 数式処理ソフトに入れてみたところ [*35], $\alpha^* = 1/96(\sqrt{5761}-65) \approx 0.114$ が最適な α となりました. ここでは最も簡単な文字のユニグラムモデルを用いましたが, 式(2.51)のトライグラム確率のような, より複雑なモデルでも数値計算になるものの, 基本的な方法は同様です. 上のように, データのうち検証のために残しておいたデータを**検証データ** (validation data) あるいは**開発データ** (development data), 統計モデルの計算に用いる, それ以外のデータを**学習データ**あるいは訓練データ (training data) といいます [20]. 検証データとしては, 一般にデータの 1 割から 2 割程度をランダムに抽出して使い, 残りの 8 割から 9 割を学習データとすることがよく行われます. 検証データが 2 割以下なのは, それより多くすると, それだけ使える学習データが減ってしまうことになるからです.

上の例ではデータの学習データと検証データへの分割を固定してしまいまし

∗35　*Mathematica* を使えば, `f[x_] := 2 Log[x]+2 Log[x+1]+Log[x+2]-5 Log[12+87 x]; Solve[D[f[x],x]==0]` と入力すれば求められます.

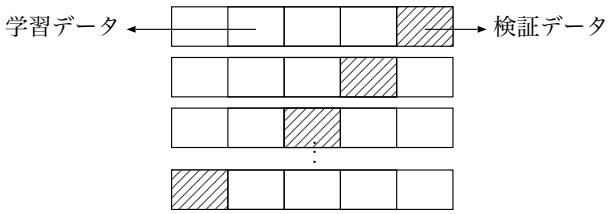


図 2.16: K 分割クロスバリデーションの様子. データを K 個 (ここでは $K=5$) に等分し, そのうち 1 つを検証データ, 残りの $K-1$ 個を学習データとする計算を K 通り行い, 結果を平均して最終的な答えとします.

たが, これはもちろん, 本当は望ましくありません. たまたま検証データに選ばれたものによって, 値が変わってしまうからです. より正確には, 上のような分割をランダムに繰り返し, 得られた α^* の平均を最終的な α^* とするのがよいでしょう. ただし, これは統計的には正しいのですが, 分割を何回繰り返せばよいかについての指針がありません. そこで, より計画的な方法として, **クロスバリデーション** (**交差検証**) という方法が知られています.

クロスバリデーション クロスバリデーションの中で最も一般的な K 分割交差検証 (K-fold cross validation) では, 検証データを毎回ランダムに選ぶかわりに, まずデータ全体を**ランダムに**, つまりシャッフルしてから K 等分します. 図 2.16 に示したように, このうち 1 つを検証データ, $K-1$ 個を学習データとする計算を K 通り行い, 結果を平均して答えとします. 上でふれたように検証データはデータ全体の 1 割から 2 割を使うのが普通ですから, K は一般に, 5 から 10 程度の値とするのがよいでしょう. K をあまり大きくすると, クロスバリデーションに必要な K 通りの計算量が非常に大きくなってしまいます.

いずれの場合でも重要なのは, 検証データをランダムに選ぶことです. たとえば, テキストが新聞記事や SNS への投稿などで時間順に並んでいる場合, 新語がある時期から初めて現れているならば, それより古い学習データからその新語を予測することは不可能です. また逆に, 検証データが学習データのすぐ後の時期になっていると, その時期に流行したテキストの確率だけが高くなればよいことになり, これも不公平になってしまいます. 統計的には, 学習データと検

証データが「同じ分布からサンプリングされた」といえる状況にしておく必要があります.

これには, Python であれば `numpy.random.permutation()` で N 個のデータについて $1, \cdots, N$ をランダムに並び換えた順番をデータ処理の際に生成してもよいですし, テキストの行をランダムにシャッフルする `shuf` のようなコマンド [*36] を使えば,

```
% shuf alice.full.txt > alice.shuffled.txt
```

として, 最初からデータの行をランダムに入れ替えたテキストを作ることができます. これは, 特に上記のクロスバリデーションなどの際には便利でしょう.

こうして学習データとは別に検証データを準備すると, 図 2.15 のように, 検証データの確率が大きくなるパラメータを求めることが可能になります.

それでは, 実際に試してみましょう. サポートサイトの `kfold-alpha.py` を上の `alice.shuffled.txt` に対して

```
% kfold-alpha.py 5 alice.shuffled.txt 1 0.5 0.1 0.01 0.001
```

のように実行すると, テキストを K 等分 (ここでは 5 等分) したクロスバリデーションを行い, 図 2.16 で検証データになったテキスト全体の確率の対数を文字バイグラムで計算して出力します. 数学的には, この後で説明するように, テキストの確率を文の確率の積だとするとき, テキストを $D_1, D_2, \cdots, D_5$ に 5 等分したとき,

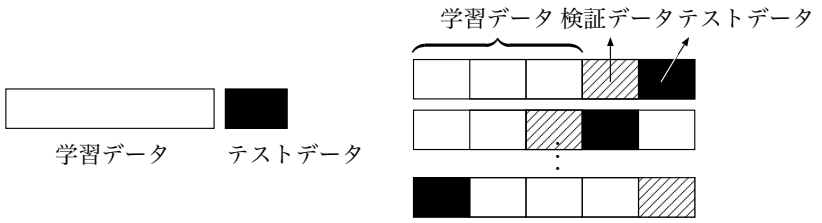
(2.55)

$$\log p(D_1|D_2, D_3, D_4, D_5) + \log p(D_2|D_1, D_3, D_4, D_5) + \cdots + \log p(D_5|D_1, D_2, D_3, D_4)$$

を, 平滑化係数 $\alpha = 1, 0.5, 0.1, 0.01, 0.001$ のそれぞれの場合で計算します. 結果は, 次のようになりました. () の中は, すぐ後で説明する 1 文字あたりのパープレキシティを表しています.

```
alpha = 1.0000 : likelihood = -332599.66 (PPL = 10.597)
alpha = 0.5000 : likelihood = -332101.93 (PPL = 10.560)
alpha = 0.1000 : likelihood = -331775.59 (PPL = 10.535)
alpha = 0.0100 : likelihood = -331866.05 (PPL = 10.542)
alpha = 0.0010 : likelihood = -332058.43 (PPL = 10.557)
```

*36　Linux では標準で含まれているようです. Mac では `% brew install coreutils` として, Homebrew で `coreutils` をインストールすると使えるようになります.



(a) 学習データと分けられたテストデータ (b) K 分割交差検証でのテストデータ

図 2.17: データの学習データとテストデータへの分割.

この結果から, `alice.txt` に対する文字バイグラム言語モデルの平滑化係数としては, $\alpha=0.1$ 程度とするのがよいということがわかります. クロスバリデーションでは値を直接最適化することはできませんので, もし, もっと細かく求めたい場合は, 候補を $\alpha=0.01, 0.05, 0.2$ などとしてクロスバリデーションをもう一度実行する必要があることに注意してください [*37]. また, パラメータが複数ある場合は, その組み合わせの数だけ計算を実行する必要があります.

なお, 式(2.55)のように検証データの確率を計算することは, 学習データに基づいて検証データを**予測** (predict) していることになりますが, これは必ずしも予測することが目的というわけではありません. たとえば人文系で, 手元のデータの解析が目的で新しいデータを解析する必要はないとしても, モデルが学習データを丸覚えした一種のトートロジー (同語反復) になっていることを避け, 統計的に正しくデータを記述していることを保証したり, α のようなパラメータを客観的に決めるためには, 検証データを分けて予測確率を計算することが必要になります.

学習データ・テストデータ・検証データ

検証データを学習データと分けておくことで, 平滑化係数 α のような統計モデルのパラメータを大まかに推定することがわかりましたが, これが真にモデルの性能を表しているとは限りません. というのは, 検証データという「正解」を見て α の値を決めているので, 厳密に言うと, これは一種のチートになってい

*37 ベイズ最適化[21]とよばれる方法を使うと, 適切な候補を見つけて計算を次々と行うことで, こうした探索を自動化することができます.

るからです. たとえば,「統計モデル」として検証データを丸覚えして確率 1 で次の単語を予測してしまえば, これは性能が最大になってしまいます.

よって, モデルを真に評価するためには, 学習時には見なかった新しいデータでの確率を計算する必要があります. 学習データと区別されたこのデータを, **テストデータ** (test data) といいます. 実は, 次の 3 章で説明するように, α のような統計モデルのパラメータを決めるには必ずしもクロスバリデーションや検証データが必要なわけではなく, ベイズ統計の枠組みでは, 学習データだけから一度の計算で α のようなパラメータの最適な値を決めることができます. よって, 学習データ–テストデータを区別することが最も本質的で, 検証データはパラメータ推定のために, 学習データの中を分けて人工的に作り出したものだといってもよいでしょう.

テストデータも, 検証データと同様にランダムに選ぶ必要があります. クロスバリデーションを使わない場合は, 図 2.17(a) のようにたとえばデータのうちランダムな 1 割をテストデータ, 残りの 9 割を学習データとするなどすればよいでしょうこれまでに見たように, 9 割の学習データの中でクロスバリデーションを行ってパラメータを決めることも可能です. 徹底的に行うには, 図 2.17(b) のように, K 分割交差検証の枠組みを使って K 個のうち 1 個をテストデータ, 1 個を検証データ, 残りの $(K-2)$ 個を学習データとする組み合わせを K 回行うことも考えられます.

いずれの場合も, モデルを学習する際には**テストデータは見ない**ことがもっとも重要です. テストデータを先に見てしまえば, いくらでもチートが可能になってしまうからです. 逆に, テストデータさえ見なければ学習データの中では何をしてもよく, K 分割交差検証は, 学習データの中で, できるだけテストデータの予測に近い状況を作り出すための一つの方法といえます.[*38]

2.6.2　予測確率とパープレキシティ

テキストの予測確率は, 学習した統計モデルから先ほどのように計算することができます. テキスト D が文 (本章では文字列) の集合

---

*38　学習に使われなかったテストデータの中には, 見たことのない文字や単語が含まれている可能性も高く, 自然言語処理では, その場合にどうするかを常に考慮する必要があります.

$$D = \{s_1, s_2, \cdots, s_N\} \tag{2.56}$$

からなっているとき, 各文が独立であると仮定すると, テキスト D 全体の確率は

$$p(D) = \prod_{n=1}^{N} p(s_n) \tag{2.57}$$

です. 文字の系列からなる文 $s_n = c_1 c_2 \cdots c_T$ の確率 $p(s_n)$ は, 式(2.45)や式(2.50)で示したように

$$p(s_n) = \prod_{t=1}^{T} p(c_t | c_1, \cdots, c_{t-1}) \tag{2.58}$$

として計算できます. ここで c_0 および c_T は 2.5.1 節で登場した, 文頭および文末を表す特殊文字 BOS および EOS だとします.

この確率を, たとえば異なる α を用いた場合で比べればよいのですが, 式(2.57)から計算されるテキストの確率は, 一般に非常に小さな値になることに注意してください. たとえば式(2.50)のような n グラム確率が非常に大きく, それぞれ $1/10 = 0.1$ だったとしても, 20 文字からなる文の確率は

$$p(s) = \left(\frac{1}{10}\right)^{20} = 0.00000000000000000001 \tag{2.59}$$

になってしまい, あっという間に計算機で表現できなくなってしまいます. そこで, この確率の対数をとれば

$$\log p(s) = 20 \log \frac{1}{10} = -46.05 \tag{2.60}$$

となり, 問題なく表すことができます. 関数 $y = \log x$ は図 2.18 のように単調増加関数ですから, $p(s)$ の大小と $\log p(s)$ の大小は一致するため, 比較にも使うことができるわけです.

ただし, このテキストの対数確率はテキストの長さに依存するため, 異なるテキストを用いて比較する場合や, クロスバリデーションでも正確に K 等分できず, テキストの長さが等しくない場合には正しい比較が行えなくなってしまいます. そこで実際には, 式(2.60)を文の長さ T で割って

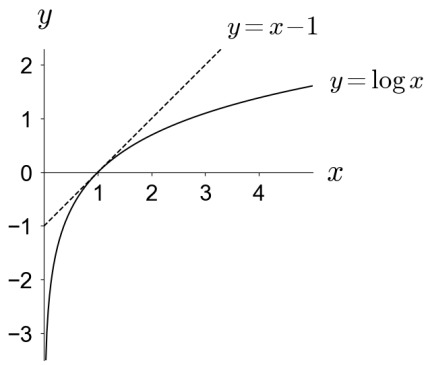


図 2.18: 対数関数 $y = \log x$ のグラフ. 点線で表したのは $x = 1$ での接線 $y = x - 1$ で, $\log x$ はつねにこの接線の下にあり, $\log x \leq x - 1$ が成り立っています.

$$\frac{1}{T} \log p(s) = \frac{1}{T} \sum_{t=1}^{T} \log p(c_t | c_1, \cdots, c_{t-1}) \tag{2.61}$$

として, 1 文字あたりの確率の対数を計算すれば, 異なるテストデータでも問題なく比べることができます. 上の例では,

$$\frac{1}{20} \log p(s) = \frac{1}{20} \cdot 20 \log \frac{1}{10} = -2.303$$

が 1 文字あたりの確率の対数になります. なお, 式(2.61)は $\log p(s)^{1/T}$ のことですから, これはテストデータの各単語の確率の**幾何平均**を計算しており, それを対数で表示している, ということになります.

2.6.3　情報理論の基礎

式(2.58)で用いた, 確率の対数

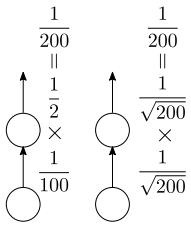
$$\log p(c_t | c_1, \cdots, c_{t-1}) \tag{2.62}$$

は, 単に計算の利便性だけでなく, 情報理論において**情報量**という意味を持っています. このことについて, 少し説明しましょう.

メモ：確率の平均について

統計モデルや機械学習では式(2.58)のように同時確率を積に分解することがよくありますが, こうした確率を平均する際には注意が必要です. [*1]

たとえば, $p(x,y) = p(y|x)p(x)$ と分解して, $p_1 = p(y|x) = 0.01$, $p_2 = p(x) = 0.5$ だったとしましょう. このとき, 単純な**算術平均**を使って片方あたりの確率を計算すると $(p_1+p_2)/2 = (0.01+0.5)/2 = 0.255$ となりますが, 言うまでもなく, これは誤りです. 図に示したように, p_1 は確率 $0.01 = 1/100$ で正解すること, p_2 は確率 $0.5 = 1/2$ で正解することを表しているので, この両方を正解する確率は $1/200$ です. よって片方あたりの正解率は, $1/\sqrt{200} = 0.071$ となるべきでしょう. もし平均を 0.255 としてしまうと, 両方正解する確率は $0.255^2 = 0.06$ となり, 真の正解率 0.005 の 10 倍以上になってしまいます. 上の計算は,



$$\sqrt{p_1 p_2} = p_1^{1/2} p_2^{1/2}$$

と**幾何平均**を考えていることになります. 確率は何かの量を表す示量変数ではなく, 「率」を表す示強変数ですから, 平均を取る場合には注意が必要です. ただし, 確率 p について自己情報量 $-\log p$ を考えれば, これは示量変数ですので, 算術平均を計算することは問題ありません. このとき,

$$\frac{(-\log p_1) + (-\log p_2)}{2} = -\frac{1}{2}(\log p_1 + \log p_2) = -\log(p_1^{1/2} p_2^{1/2})$$

ですから, 結局これは確率の幾何平均を計算してから, 自己情報量に戻していることになります.

たとえば, 可能な選択肢が 8 つあってどれも等価なとき, どれかが選ばれる確率は $1/8$ です. つまり逆にいうと, 確率 $1/8 = 0.125$ とは, 等価な選択肢が 8 つある状況と同じであることを意味しているわけです. この等価な選択肢の数のことを, **分岐数**といいます.

---

*1 同時確率とはみなせない, 複数のモデルによる確率のような場合は, 算術平均をとっても問題ありません.

このとき実際に選択肢から 1 つを選ぶのに, 8 回の選択をする必要はありません. 8 個の選択肢を $x_1, x_2, \cdots, x_8$ と表すと, コインを投げて表裏のどちらが出たかを使うことにして,

- 1 回目のコインで $(x_1, \cdots, x_4)$ と $(x_5, \cdots, x_8)$ のどちらを選ぶかを決め, 前者の $(x_1, \cdots, x_4)$ が選ばれたとすると
- 2 回目のコインで (x_1, x_2) か (x_3, x_4) のどちらを選ぶかを決め, 後者の (x_3, x_4) が選ばれたとすると
- 3 回目のコインで x_3 と x_4 のどちらを選ぶかを決め, ここでは表が出たので, 最終的に x_3 を選ぶ

のようにすれば, どれかをランダムに選ぶことができます. このように, 8 個の選択肢であれば $\log_2 8 = 3$ 回のコインを振れば出力を決めることができるため, 必要な情報量は 3 だということができます. 同様にして, 確率 p の事象は $\dfrac{1}{p}$ 個の等価な選択肢から 1 つを選ぶことと等価で, このときの手数, すなわち必要な情報量は $\log \dfrac{1}{p}$ です. そこで, 確率 p の事象の「**情報量**」を表すこの量を Shannon の**自己情報量** (self information) といい,

$$I(x) = \log \frac{1}{p(x)} = -\log p(x) \qquad (\textbf{自己情報量}) \tag{2.63}$$

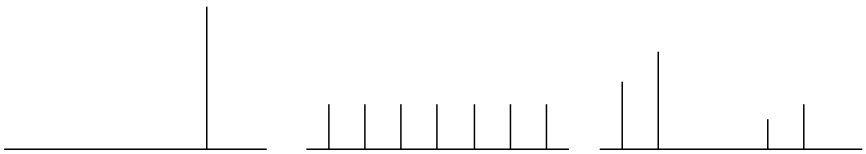
と書きます. 対数の底は任意ですが, 上のように 0/1 の決定をもとに底を 2 にする場合は bit (binary digit の略), 自然科学や計算機上の対数で通常用いられている e を底にする場合は nat (natural digit) が単位となります. [*39] たとえば, 確率 $p(x) = 0.07$ の単語が出現したとき, これは $1/0.07 = 14.3$ 個の等価な選択肢から 1 つが選ばれたことと同じなので, その情報量は

$$I(x) = \log \frac{1}{0.07} = \log 14.3 = 2.66 \text{ (nat)} \tag{2.64}$$

になります. 通常は対数の底は一定に揃えて考えているため, 単位は省略するの

---

*39　1000 個の選択肢があるとき, その情報量は自然対数を使えば $\log 1000 = 6.91$ (nat) です. 一方で仮想的に 1000 面サイコロを準備して, このサイコロで選ぶ手数を kit と定義すれば, この情報量は 1(kit) になりますが, 1000 面のサイコロを振って結果を求めるための計算量は別にかかりますから, 底を大きくすれば情報量が減るわけではなく, 値のスケールは対数の底に依存します.



(a) エントロピー最小の $p(x)$. $H = 0$　(b) エントロピー最大の $p(x)$. $H = \log K = 1.95$　(c) 中間的な分布 $p(x)$. $H = 1.28$

図 2.19: 確率分布 $p(x)$ とそのエントロピー $H = -\sum_x p(x) \log p(x)$.

が一般的です. 本書では特に断りのない限り, 対数としては自然対数を用います.

メモ：符号長について

底となる数を d としたとき, 情報量とは, 対応する分岐数を d 進法で表したときに何桁になるか, すなわち, 何個の数字が必要になるかという量だといってもよいでしょう. これを**符号長**とよび, 式(2.63)の情報量とは符号長とみなすことができます. たとえば確率 $1/16 = 0.0625$ は分岐数 16 に対応しますから, これは 2 進数では 1011 のような 4 桁の数で表され, 情報量は 4 (bit) になります. nat で情報量を考える場合は, 仮想的に $e \simeq 2.72$ 進数を使っていると考えればよいでしょう.
このように情報理論と情報圧縮には密接な関係があり, 可能なデータに高い確率を与えること, すなわち, よいモデリングを行うことと, それを短い符号長で表すことは, ほぼ等価な問題です[22][23].

われわれが実際に扱う確率は多くの場合, 単一の確率 $p(x)$ というより, アルファベット $\mathcal{X}$ 上の確率分布 $\{\, p(x) \,|\, x \in \mathcal{X} \,\}$ でしょう. このとき, 式(2.63)で表される x の自己情報量を, $p(x)$ で期待値をとった

$$H = \sum_x p(x)(-\log p(x)) = -\sum_x p(x) \log p(x) \tag{2.65}$$

を, 確率分布 $p(x)$ の**エントロピー** (entropy) とよびます. よってエントロピーとは, 確率分布 $p(x)$ から現れる記号のもつ情報量の期待値で,

$$H = -\sum_x p(x) \log p(x) = \big\langle -\log p(x) \big\rangle_{p(x)} \tag{2.66}$$

とも書くことができます. $\langle\cdots\rangle_{p(x)}$ は $p(x)$ で期待値をとる, すなわち $\mathbb{E}_{p(x)}[\cdots]$ の略記法で, 本書でも以下必要に応じてこの表記を用います. 図 2.19 に示したように, $p(x)$ がたとえば $(0,1,0,\cdots,0)$ のようにどれかの確率が 1 で他が 0 のときにエントロピーは最小となり, 式(2.65)から $H = 1\cdot(-\log 1) = 0$ です.[*40] いっぽう, $p(x)$ が一様分布 $(1/K, 1/K, \cdots, 1/K)$ のときエントロピーは最大になり, $H = -K\cdot\dfrac{1}{K}\log\dfrac{1}{K} = \log K$ が最大値となります.

これからわかるように, エントロピーは確率分布 $p(x)$ の「**曖昧さ**」を示す量となっています. 最も曖昧な一様分布では次に何が来るかについてまったく予想できませんから, エントロピーは選択肢の数の対数で $\log K$, 曖昧性がなく結果が完全に予想できる場合は結果は 1 通りということですから, エントロピーは $\log 1 = 0$ となるわけです. 符号長の言葉で言えば, エントロピーとは, 確率分布 $p(x)$ から現れる記号を符号化するのに必要な符号長 (= 情報量) の期待値と考えることができます.

パープレキシティ　式(2.61)で計算したテキストの平均的な予測確率の対数, すなわち (負の) 情報量

$$\frac{1}{T}\log p(s) = \frac{1}{T}\sum_{t=1}^{T}\log p(c_t|c_1,\cdots,c_{t-1}) \tag{2.67}$$

は, よいモデルであるほど予測確率が高いため, 大きな値になります. ただしこの値は, 対数の底に依存することや, 対数をとっているために差がわかりにくいという問題があります.

たとえば平均予測確率が 0.01 のモデルを改善して, 2 倍の 0.02 にしたとしても, 平均予測確率の対数は $\log 0.01 = -4.61$ から $\log 0.02 = -3.91$ になるだけで, 差は 0.69 しかありません. この値の解釈も難しいため, モデルの評価には, 平均予測確率の逆数, すなわち分岐数を用いることがよく行われます. これを**パープレキシティ**(perplexity) といいます.[*41] すなわち, パープレキシティとは**平均分岐数**のことです. 例えばいまの例では, パープレキシティが 1/0.01＝100

∗40　なお, $0\log 0 = 0$ と約束します.

∗41　perplex は “まごつかせる” という意味で, これはモデルが次の単語を選ぶのにどれだけ「まごつく」のか, ということを表しています.

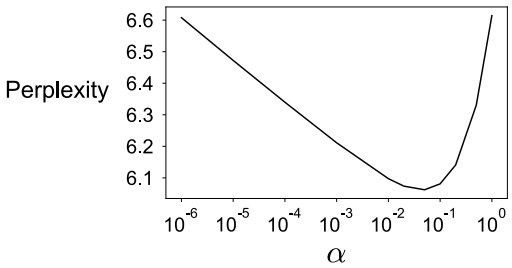


図 2.20: テストデータ `alice.test.txt` のパープレキシティと, 文字トライグラム言語モデルの α の値の関係. 横軸は対数軸になっていることに注意してください.

から $1/0.02 = 50$ になった, と考えるわけです. この値は直感的で, 対数の底に依存しません. パープレキシティは (幾何) 平均予測確率の逆数ですから, 実際には式(2.61)から, 次のようにして計算します.

(2.68)

$$\begin{aligned}\mathrm{PPL}(s) &= 1 \Big/ \left(\prod_{t=1}^{T} p(c_t|c_1,\cdots,c_{t-1})\right)^{1/T} = \left(\prod_{t=1}^{T} p(c_t|c_1,\cdots,c_{t-1})\right)^{-1/T} \\ &= \exp\left(\log\left(\prod_{t=1}^{T} p(c_t|c_1,\cdots,c_{t-1})\right)^{-1/T}\right) \\ &= \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log p(c_t|c_1,\cdots,c_{t-1})\right). \qquad (\textbf{パープレキシティ})\end{aligned}$$

パープレキシティは小さいほど予測確率が高い, よいモデルであることを意味します. 最小値はすべての確率が 1 のときですから $1/1 = 1$, 最大値はすべての確率が $1/V$ (V は語彙の数) の等確率の場合で V になります. よって, パープレキシティは言語の場合は, **語彙の大きさによってスケールが異なる**ことに注意してください.[*42] 図 2.20 に, 異なる α の値について `alice.txt` のテストデー

*42 パープレキシティが語彙の大きさに依存するため, 別のテキストと比べられないというこの欠点を解消するため, 最近, PPLu という新しい指標が提案されました[24]. PPLu では, 式(2.68)で予測確率 $p(c_t|c_1,\cdots,c_{t-1})$ ではなく, そのユニグラム確率との比 $p(c_t|c_1,\cdots,c_{t-1})/p(c_t)$ の積を計算します. これは相対的な値であるため語彙の大きさによらず, 3.2.2 節の議論から文脈 $c_1,\cdots,c_{t-1}$

タで計算したパープレキシティの値を示しました. パープレキシティはテキストの予測確率から計算されるため, 図 2.15 と本質的に同じ意味を持っていますが, 1 単位 (ここでは 1 文字) あたりの値であるため, テキストの長さによらない指標になっているという特徴があります.

ところで, 実際には通常のテキストにおいて, パープレキシティ1, すなわちすべての予測確率が 1 の「完璧なモデル」を達成することは原理的に不可能で, ある**下限**が存在します. これはなぜでしょうか. そして, その下限は何を表しているのでしょうか?

エントロピーとクロスエントロピー

いま, 言語 $\mathcal{L}$ に含まれるすべての要素 $x \in \mathcal{L}$ について, 自然のもつ真の確率モデル $p(x)$ を, 統計モデル $q(x)$ を使って近似することを考えましょう. ここで要素 x として考える単位は文字や文, 文書など何でもかまいません. $q(x)$ をできるだけ $p(x)$ に近づけたいのですが, こうした確率分布の間の距離を測るのに, 次の**Kullback**(カルバック)**–Leibler**(ライブラー)**ダイバージェンス** (**KL ダイバージェンス**) という基本的な量があります. [*43]

$$D(p||q) = \sum_{x \in \mathcal{L}} p(x) \log \frac{p(x)}{q(x)} \tag{2.69}$$

KL ダイバージェンス $D(p||q)$[*44] は $p = q$ のときに最小値 0 をとり, 常に非負の値をとる, すなわち

$$D(p||q) \geq 0 \qquad (\textbf{KL ダイバージェンスの非負性}) \tag{2.70}$$

をみたします. これは, 次のようにして示すことができます. 図 2.18 に示したように, $y = x - 1$ は $y = \log x$ の接線となり, 必ず $\log x \leq x - 1$ ですから,

---

と次の語 c_t の自己相互情報量という意味を持っており, 実験的にも有効な指標であることが確かめられています. 詳しくは, 原論文[24]を参照してください.

*43　Solomon Kullback (1907–1994) と Richard Leibler (1914–2003) はアメリカの数学者・暗号研究者です. KL ダイバージェンスは, 1951 年の論文[25]で導入されました. 世界初のコンピュータ ENIAC の登場が 1946 年だったことに注意してください.

*44　KL($p||q$) と書くこともあります.

$$\sum_x p(x)\log\frac{q(x)}{p(x)} \le \sum_x p(x)\left(\frac{q(x)}{p(x)}-1\right) = \sum_x q(x) - 1 = 0 \tag{2.71}$$

が成り立ちます. 両辺に -1 をかければ, すべての p, q について

$$\sum_x p(x)\log\frac{p(x)}{q(x)} \ge 0 \tag{2.72}$$

が得られます. □

KL ダイバージェンス $D(p||q)$ は, p と q について**対称ではない**ことに注意してください.[*45] 実際, 式(2.69)を書き直すと

$$\sum_{x\in\mathcal{L}} p(x)\log\frac{p(x)}{q(x)} = \left(\sum_{x\in\mathcal{L}} p(x)(-\log q(x))\right) - \left(\sum_{x\in\mathcal{L}} p(x)(-\log p(x))\right) = \big\langle -\log q(x)\big\rangle_{p(x)} - \big\langle -\log p(x)\big\rangle_{p(x)} \tag{2.73}$$

ですから, KL ダイバージェンスは前節の議論から, 言語 $\mathcal{L}$ のテキストを「近似モデル q で符号化したときの符号長」と「真のモデル p で符号化したときの符号長」の差の期待値を計算していることになります. [*46] KL ダイバージェンスが非負であるということは, 符号化の意味では, 真のモデル p より, 近似モデル q で符号化した方が平均的には必ず長い符号を必要とする, ということを意味しています.

この KL ダイバージェンスを使うと, われわれは言語の持つ真の $p(x)$ を統計モデル $q(x)$ で近似しようとしているのですから, その差は

---

*45 このため, $D(p||q)$ は距離の公理を満たさず, 厳密には通常の意味での「距離」ではありません. ただし, $D(p||(p+q)/2)$ と $D(q||(p+q)/2)$ の平均をとることで対称性を持つ **Jensen-Shannon ダイバージェンス**というダイバージェンスも存在し, 様々な研究で使われています. 詳しくは, この後で紹介する情報理論の教科書を参照してください.

*46 または, 「真のモデル p を統計モデル q で近似したときに, 平均的にどれだけビット落ちするか」を計算しているといってもいいでしょう.

$$D(p||q) = \sum_{x\in\mathcal{L}} p(x)\log\frac{p(x)}{q(x)} = -\sum_{x\in\mathcal{L}} p(x)\log q(x) - \left(-\sum_{x\in\mathcal{L}} p(x)\log p(x)\right) \geq 0 \tag{2.74}$$

です. これから,

$$-\sum_{x\in\mathcal{L}} p(x)\log q(x) \geq -\sum_{x\in\mathcal{L}} p(x)\log p(x) = H(p) \tag{2.75}$$

が成り立つことがわかります. 式(2.75)の右辺は言語の持つ真の確率分布 $p(x)$ のエントロピー $H(p)$ で, これは定数です. いっぽう, 左辺は確率分布 p と q の **クロスエントロピー** (cross entropy) とよばれる量

$$H(p,q) = -\sum_{x\in\mathcal{L}} p(x)\log q(x) \qquad (\textbf{クロスエントロピー}) \tag{2.76}$$

で, 言語のあらゆる文 x について, 統計モデル $q(x)$ で計算した情報量 $-\log q(x)$ の, 真の確率分布 $p(x)$ についての期待値です. すなわち, 式(2.75)は, あらゆる確率分布 p, q について, p に関する q のクロスエントロピーは p のエントロピーより大きく,

$$H(p,q) \geq H(p) \qquad (\textbf{クロスエントロピーとエントロピー}) \tag{2.77}$$

であることを表しています. また, 言語の持つ真の確率分布 p のエントロピー $H(p)$ は定数ですから, 式(2.74)より, クロスエントロピーの最小化は, KL ダイバージェンスの最小化と等価であることがわかります.

いま, 手元の N 個の $x_1, x_2, \cdots, x_N$ について, N が充分に大きければ, これは言語の持つ真の確率分布 $p(x)$ からのサンプルだと考えることができますから, 式(2.75)および式(2.77)の左辺のクロスエントロピーは

$$-\sum_{x\in L} p(x)\log q(x) \simeq -\frac{1}{N}\sum_{n=1}^{N}\log q(x_n) \tag{2.78}$$

とモンテカルロ近似することができます.[*47] 式(2.78)を式(2.75)の左辺に代入

*47　関数 $f(x)$ の確率分布 $p(x)$ に関する期待値 $\int p(x)f(x)dx$ を解析的に計算するのが難しいと

して, 両辺を指数の肩にのせれば, [*48]

$$\mathrm{PPL} = \exp\left(-\frac{1}{N}\sum_{n=1}^{N}\log q(x_n)\right) \geq e^{H(p)} \tag{2.79}$$

が得られます. 式(2.79)の左辺は式(2.68)のパープレキシティですから, **パープレキシティには超えられない下限が存在し**, それは言語の持つ真の確率分布 $p(x)$ のエントロピー $H(p)$ を用いて $e^{H(p)}$ と書けることになります.

$e^{H(p)}$ は言語の持つ真の平均分岐数 (われわれには未知) で, **その言語の複雑さ**を表しています. よって式(2.79)は, 統計モデルのパープレキシティは決して言語の持つ真の平均分岐数より小さくはできない, ということを意味しています.

これは, 簡単な人工例を考えてみればわかりやすいでしょう. たとえば, 0 と 1 が完全にランダムに出現する

11000100111010101000000011…

のような系列にどんな統計モデルを考えても, もとの系列がランダムなので, 決して次を予測することはできません. この場合, 語彙は 0 と 1 の 2 つなので, パープレキシティの最小値 (および最大値) は 2 になります.

一方, 0 と 1 がそれぞれ確率 $(0.9, 0.1)$ で現れる

00000110000001000000001000…

のような系列の場合, 真のモデルと等しい $(q(0), q(1)) = (0.9, 0.1)$ という統計モデルを推定することで, パープレキシティを 2 よりずっと下げることができます. しかし, それはエントロピー $H(p) = -0.9\log 0.9 - 0.1\log 0.1 = 0.325$ から得られる $e^{H(p)} = 1.38$ より小さくすることはできません. というのは, 正しいモデルを知っていても, 次の数字が 0 か 1 かには常にランダム性が残っており, 次

き, $p(x)$ からランダムにサンプリングされた N 個の $x^{(i)}$ $(i = 1, \cdots, N)$ を使って, $\int p(x)f(x)dx \simeq \frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})$ と数値的に積分を近似することをモンテカルロ積分とよびます. 詳しくは, 機械学習におけるモンテカルロ法の優れたチュートリアルである[26]や, 教科書[23]の 29 章を参照してください. 情報理論では, エルゴード情報源について Shannon-McMillan-Breiman の定理とよばれる定理でこの置換を行います[27].

*48 $y = e^x$ は単調増加関数なので, $a < b$ であることと $e^a < e^b$ であることは等価です.

の数字を完全に正確に予測することは原理的に不可能だからです. これが, パープレキシティに情報理論的な下限が存在する理由です.

言語のエントロピー　なお, 情報理論を創始した Shannon は, 1948 年の最初の論文[28] の中で, 文字の n グラムモデルなど様々な方法を用いて英語のエントロピーの上限を計算し, 1 文字あたりほぼ 1 ビットという結果を得ています. Brown コーパスを使って計算すると [*49], 英語の単語の平均的な長さは約 4.7 文字ですから, これは 1 単語あたりのパープレキシティでは $2^{4.7} = 26$ に相当します.

現代の大規模なニューラル言語モデルを使うとさらに正確な推定が可能になり, 5000 億語 [*50] のコーパスから学習された, 執筆時点で最大の超巨大な言語モデル GPT-3 [29]では, Penn Treebank コーパスにおいて単語あたりのパープレキシティ20.5 が得られたことが報告されています. 式(2.79)から, 言語の真のエントロピーは $\log 20.5$ より小さいと考えられますが, これまでの議論から, 原理的にそれが 0 になることはありません. [*51]

2.6.4　統計モデルと汎化性能

これまでに行ってきた, 学習データによる統計モデルの学習とテストデータによる評価を, もう一段高い視点から見てみることにしましょう.

図 2.21 に示したように, (教師なし学習の) 統計モデルとは, あらゆる可能なデータ $D \in \mathcal{D}$ について, 真の確率 $p(D)$ の推定値である確率 $q(D)$ を与える確率分布 $\{q(D)\}_{D\in\mathcal{D}}$ と考えることができます. たとえば, D として「瓶銅肩果扉暴テ秋銀露呂非…」のようなテキストには低い確率を, 「こんやの星祭に青いあかり…」のようなテキストには高い確率を与えるのが, 良い統計モデル $q(D)$ というわけです.

---

*49　awk を使うと, `% awk '{for(i=1;i<=NF;i++){s+=length($i);n++}};END{print s/n}' brown.txt` とすれば簡単に求めることができます.

*50　正確には, これは単語を細分した Byte-pair トークンですので, 実際の単語数よりは若干大きくなっています.

*51　鋭い方は, これが「ラプラスの魔」と同じ問題であることに気がつかれたでしょう. 偶然性や自由意志を否定し, すべての機構を知れば世界は完全に確定的に動いていると考えればエントロピーの下限は 0 になりますが, 仮にそうだとしても, 有限時間を生きる人間がその機構を完全に知ることができるかは別の問題で, 統計的に推定するしかないとも考えられます.

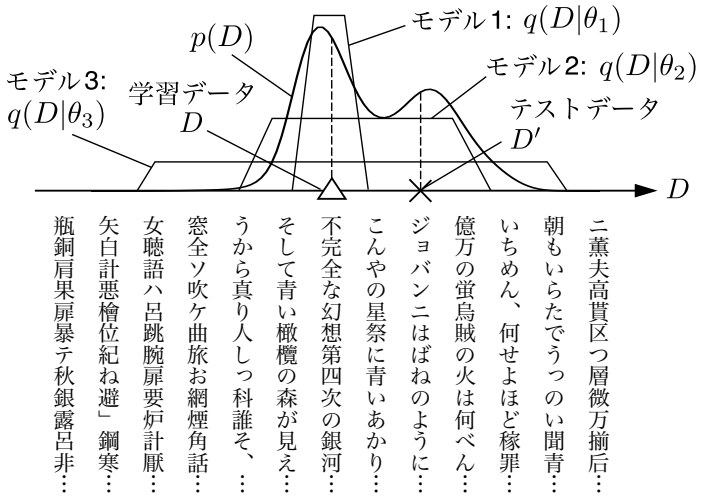


図 2.21: 統計モデルの過学習と汎化性能. モデル 1 は与えられた学習データ D (△印) に過学習してしまい, テストデータ D' (×印) に非常に低い確率しか与えられなくなっています. モデル 3 は一般的すぎて 逆に過少適合となるため, テストデータに一番高い確率を与えるのは, 真のデータ分布 $p(D)$ に近く, 汎化性能が高いモデル 2 となっています. 図の下に, 『銀河鉄道の夜』を学習データとした場合の可能なデータ空間の例を示しました.

いま, 手元にある学習データ D および テストデータ D' はどちらも, 真の分布 $p(D)$ からのサンプルだと考えられますが, 一般に $D \neq D'$ なので, 図 2.21 のモデル 1 のようにあまりに現在の D の周りに特化した確率分布 $q(D)$ を学習してしまうと, D とは違う D' での確率 $q(D')$ が大きく下がってしまいます. これを**過学習** (**オーバーフィット**) といいます. よって, 統計モデルの学習の目標とは, 学習データ D から学習しつつ, 未知のテストデータ D' をうまく予測できる, すなわち D' の確率 $q(D')$ が高くなるように学習を行うことです. こうした, テストデータにおける性能を**汎化性能** (genelization ability) といいます. われわれはあらゆる可能なテキストを観測するわけにはいきませんが, 学習データを用いて, できる限り汎化性能の高い (すなわち, 真の分布 $p(D)$ に近い) 確率モデル $q(D)$ を学習したい, ということになります.

もちろん, 図 2.21 のモデル 3 のように可能なデータ全体を薄くカバーする確

率モデルを推定すれば, どんなテストデータにも一定の確率を与えることができます. たとえば 2.1 節の文字ユニグラムモデルを用いれば, 「うから真り人しっ科誰そ」のような意味不明な日本語にも, 0 でない確率が与えられるわけです. しかし, これは明らかにモデルが一般的すぎますから (これを**過少適合** (**アンダーフィット**) といいます), 最もよいのは図 2.21 のモデル 2 のように, 学習データもテストデータも適度にカバーする, 真の分布 $p(D)$ に近いモデルを見つけることです. 実際このとき, テストデータにおける確率は図からわかる通り, モデル 2 によるものが最も高くなります. よって, 統計モデルの学習には検証データ D' を学習データ D 以外からランダムに選び (これは真の分布 $p(D)$ からのサンプルだと考えられます), そこでの確率 $q(D')$ が最も高くなるようにすればよい, ということになります. これを系統的に行うのが, 先に示したクロスバリデーションです.

なお, 統計モデルのパラメータ θ に事前確率 $p(\theta)$ を置くベイズ的なモデルを考えれば, 検証データを使わずに学習データだけから, 汎化性能が高い統計モデルを学習できることを示すことができます. これについては, 3 章でベイズ推定を導入した後で説明します.

2 章のまとめ

2 章では, 文字の統計モデルを例にして確率, 同時確率, 条件つき確率といった統計の基本的な概念と, 同時確率の周辺化, ベイズの定理といった操作について説明しました. 特に, 条件つき確率やベイズの定理は最も基本的な確率の連鎖則の式(2.20)からただちに導くことができますので, 公式を暗記する必要はありません.

条件つき確率を使うと n グラム言語モデルを作ることができ, 文の確率を計算したり, 逆に文を確率的に生成することができます. 文字 n グラム言語モデルは「単語」の概念すらない最も基本的なものですが, 例にみたように, かなり日本語や英語に近い文字列を生成することができます. [*52] 特に単語の綴りは, 文字 n グラムで非常によく生成することができました.

---

*52　上で述べたように, これは Shannon による情報理論の最初の論文[28]で行われている実験です.

nグラム言語モデルのような統計モデルの性能は, 学習データとテストデータを分け, 学習データで計算した統計モデルがテストデータを予測できるかで測ることができます. この性能のことを, 汎化性能と呼ぶのでした. 統計モデルを使えば, アドホックな方法とは異なり, 開発した方法をこうして客観的に評価し, 改善することができます. [*53] 現代の深層学習でも, 背後にはすべてこうした統計的な概念が存在しています.

*53 言うまでもなく, これはある方法が科学であるための要件そのものです.

ノート：テキスト処理のための言語

本書ではプログラム言語として Python を主に使用していますが, Python だけが唯一の言語というわけではありません. Python の前に一世を風靡した Perl [30]は, 非常にテキスト処理に長けた言語でした. 作者の Larry Wall はバークレー校で言語学を学んでいたため, Perl の構文には`$@&`による変数の「品詞」, `$_`による「痕跡」など, 言語学の要素が多く取り入れられ, その機能の一部は Python にも継承されています.

また, テキストを扱うのを得意とする `awk` や `sed` といった言語が, Linux や MacOS のような Unix には古くから標準的に含まれており, 利用できます.[*54] 43 ページの脚注でも使用した `awk` は, テキストを 1 行読むごとにフィールドを空白で自動的に分割して`$1`,`$2`,…という名前をつけてくれますので, たとえばテキストの `Foo` で始まる各行の 2 個目のフィールドの総和を求めたければ,

```
% awk '/^Foo/{s+=$2};END{print s}' input.txt
```

のように簡潔に書くことができ, 通常は Excel で行うような計算を簡単に行うことができます. `awk` は他にも exp, log, sin のような算術演算や `substr` のような文字列演算, `for` 文による繰り返しなども備えており, 簡単なデータ解析にもたいへん有用です.

`sed` は stream editor の略で, 1 行しか画面出力を持たないエディタ `ed` [*55] の拡張ですが, それゆえに簡潔なコマンド体系を持っています. たとえば

```
% sed '1,5d;s/foo/bar/g' input.txt
```

とすれば, 1 行目から 5 行目を削除 (`d`) した上で, `foo` をすべて (`g`)`bar` に置換 (`s`) して出力します.

∗54 これは, 歴史的に Unix がテキスト処理をその目的の一つとして開発されたためです.
∗55 `ed` については, 『The UNIX Super Text』[31]などを参照してください.

また

```
% sed G input.txt
```

とすれば, 1 行おきに空白を入れた「ダブルスペース」のテキストを簡単に作ることができます. パターンスペースやホールドスペースといった内部の記憶領域をうまく使うと, さらに高度な処理も可能で, 何と図 2.22 のように sed で書かれたテトリス [*56] やチェス [*57] すら存在します.

sed や awk については『sed&awk プログラミング』[32]『プログラミング言語 awk』[33] といった標準的教科書があるほか, 「awk の簡単な使い方」[*58] 「sed 教室」[*59] 「sed は日暮れて」[*60] といったフリーのチュートリアルがあります. こうした言語を適宜使いこなすことで, テキストを計算機でより自由に扱えるようになるでしょう.

なお, 本書を書く際に用いた組版プログラム LaTeX (TeX) も, テキストを処理するプログラミング言語の一種といえます. チューリング完全, すなわち Python のような言語と同じ表現能力を持っているため, TeX で書かれた BASIC インタプリタ BASIX[*61] [34]など, 驚くべきプログラムも存在しています.

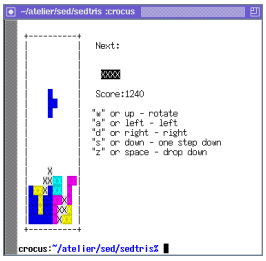


図 2.22: 文字端末で, sed で書かれたテトリスをプレイしている様子.

*56 https://github.com/uuner/sedtris
*57 https://github.com/bolknote/SedChess
*58 「awk の簡単な使い方」http://chasen.org/~daiti-m/etc/awk/ に転載.
*59 「sed 教室」https://www.gcd.org/sengoku/sedlec/
*60 「sed は日暮れて」https://chimimo.tumblr.com/post/8995558289/sed1

2 章の演習問題

(1) alice.txt や brown.txt で, j に続く確率の高い文字にはどんなものがあり, その確率はそれぞれ何でしょうか.

(2) 上のテキストで, z や th の前に来ることのできる文字にはどのようなものがあり, それらの確率は何でしょうか.

(3) brown.txt など, alice.txt 以外の英語のテキストでは, alice.txt と文字バイグラム確率にどのような違いがあるでしょうか. どのバイグラムの確率が特に異なるかを自動的に検出するには, どうすればよいでしょうか.

(4) 本章では簡単のために英語のテキストを使いましたが, 日本語のテキストの場合は, 文字のユニグラム確率やバイグラム確率はどうなっているでしょうか. 漢字を含めると文字の種類が厖大になってしまうので, ひらがなまたはカタカナだけに絞るとよいでしょう. 図 2.3 のような行列を作ると, どうなっているでしょうか.
サポートサイトの data フォルダに,『更級日記』のテキスト sarashina.txt を置きました [*62]. この場合のひらがなのユニグラム確率, バイグラム確率は, 現代の日本語と比べるとどのような違いがあるでしょうか.
Python で文字列がすべてひらがなから成っているかを判定するには, regex パッケージをインストールした後で, Unicode の文字プロパティを使って

```
import regex
def is_hiragana (s):
    return regex.search(r'^\p{Hiragana}+$', s)
```

のような関数を書くと行うことができます.

(5) 日本語のテキストで, 読点「、」の前に来る文字には, どんな傾向があるでしょうか.

(6) 上で使ったような日本語のテキストを使って, 文字 n グラムモデルからランダムに文を生成するとどうなるでしょうか.

---

*61　https://www.ctan.org/tex-archive/macros/generic/basix

*62　このテキストは, 渋谷栄一氏が http://genjiemuseum.web.fc2.com/sara2.html で公開されているものを整形して使用しました.

(7) 本章で紹介した言語モデルよりも性能のよい, 3.4.5 節の Kneser–Ney 言語モデルを使って, 4 グラムや 5 グラムの文字 n グラムから 2.5.2 節のようにテキストを生成してみると, どうなるでしょうか. 本章で紹介した 3 グラムの場合と, どんな違いがあるでしょうか.

(8) 単語は, 文字からなる短い「文」とみなすことができます. 単語辞書を使って, 43 ページのように単語の綴りをランダム生成してみましょう. どんな単語ができるでしょうか.
Linux や MacOS のような Unix では, 単語辞書は通常, `/usr/share/dict/words` に置いてあります. BOS, EOS を必ず使うように注意してください. また, このようにして生成された単語には, 現実の単語と比べてどのような問題があるでしょうか.

(9) ジェイムズ・ジョイスの小説 “Finnegan’s Wake” (1939) は, 言葉遊びに満ちており, `prumptly` や `sesthers` といった, 通常の英語では見たこともないような単語が多数登場する, 複雑な作品です. こうしたテキストを扱うには, 単語の言語モデルではなく, 文字の言語モデルを考えるのが適切でしょう. サポートサイトの `data` フォルダにある `finnegans.txt` を用いて, この小説のパープレキシティを計算してみましょう. `brown.txt` のような通常の英語と比べて, パープレキシティはどうなっているでしょうか. また, 43 ページのようにして, この作品から (ジョイス自身が行ったように)「単語」をランダムに生成すると, どうなるでしょうか.

(10) 英語や日本語以外のテキストでランダム生成してみると, どうなるでしょうか. プログラミング言語も, 一種の形式言語です [*63]. プログラムのテキストから文字 n グラム言語モデルを学習して, ランダム生成するとどうなるでしょうか. その場合, 何が問題になるでしょうか.

(11) X も Y も二値のとき, 2.4.2 節のベイズの定理を使って, Y が与えられたときの X の条件つき確率を計算する一般的なプログラム `bayes.py` を書いてみましょう. `lik.dat` および `prior.dat` をそれぞれ次のようなテキストファイルとして与え, 観測値 Y を与えると, `bayes.py` は $p(X|Y)$ を

*63 プログラミング言語のように, 人間が定義した言語である形式言語と区別する意味で, 日本語や英語のような言語は「自然言語」と呼ばれています.

計算します.

```
% cat lik.dat
1   0
0.3 0.7
% cat prior.dat
0.2 0.8
% bayes.py lik.dat prior.dat 0   (← Y=0 が観測された)
posterior: p(X|Y=0) = [0.454, 0.545]
% bayes.py lik.dat prior.dat 1   (← Y=1 が観測された)
posterior: p(X|Y=1) = [0, 1]
```

2章の文献案内

統計的自然言語処理の最も有名で基本的な教科書は, 現在も最前線で活躍する研究者の Schütze と Manning による “Foundations of Statistical Natural Language Processing” (FSNLP とよばれています) [35] でしょう. 最近になり, 日本語訳も出版されました[36]. FSNLP は 1999 年に出版されたため, 本書で説明した内容で扱われていないものも多くありますが, 自然言語処理を最も基礎から体系的に説明している教科書といえます. 本書で触れられなかった論点も網羅されているため, 自然言語処理を専門とする場合はぜひ一読することをお薦めします. 日本語の教科書としては, 北による『確率的言語モデル』[37]も同じ 1999 年の出版ながら, 今なお研究者に広く読まれている名著です. 本書のタイトルは, この本のオマージュでもあります. 本書は『確率的言語モデル』で扱われていない様々な最新の知見を盛り込み, 主に文を対象とした確率的言語モデルから, より広く文書やテキストを, その差異も含めてベイズ統計の立場から扱うものに拡張しました.

統計的自然言語処理は機械学習の一部でもあり, 2.6 節で扱ったような統計モデルの学習の一般的な話題は, Bishop による “Pattern Recognition and Machine Learning” (通称 PRML) [20]で詳しく説明されています. 筆者も翻訳に参加した日本語版[38]も出版されており, 多くの方に読まれています. また, ケンブリッジ大学の MacKay による “Information Theory, Inference, and Learning Algorithms” (ITILA) [23]は, ベイズ統計と情報理論に基づいた深い考察に支えられて機械学習を広くカバーする名著で, 全文の PDF を公式ページ [*64] からフリーでダウンロードすることができます. 本章前半の議論は, 多くこの本を参考にしました.

2.6.3 節でも説明したように, 情報理論は機械学習一般, 中でも自然言語処理とは深いつながりがあります. 多くの場合は `0/1` や `a〜z` のアルファベットの系列を扱う情報理論を, 単語や文, 文書といったより上位の構造を含めて徹底的に高度化したのが, ある意味で自然言語処理であるといってもいいでしょう. もちろん, 情報理論にはそれ以外に符号化や伝送に関わる, 多くの技術的内容が含ま

[*64] `http://www.inference.org.uk/mackay/itila/book.html`

れています. 情報理論では, Cover&Thomas の教科書[27] (和訳[39]) が最も基本的な文献として知られています. また, 上の ITILA [23] でも詳しく扱われています. 日本語では, 韓による『情報と符号化の数理』[22] は透徹した哲学に基づいて情報理論を解説した名著で, これを読むことでより本質的な理解と, 情報理論の懐の深さに触れることができるでしょう.

情報理論にもとづく定式化は深層学習時代においても見直されつつあり, 気鋭の若手 Cottrell による COLING 2022 のチュートリアル “Information Theory in Linguistics: Methods and Applications”[*65] では, 言語学的な応用も含めて幅広い研究が紹介され, 演習用の Jupyter Notebook も公開されています. また, 統計数理研究所の伊庭による「「情報」に関する 13 章」[40] [*66] は, そもそも「情報」とは何か?という根本的な疑問を抱いている方にもお薦めできる, たいへん読みやすく, かつ奥の深い論考です.

人文学においてもテキストデータの使用は以前から行われており, 2001 年の近藤らの論文[41]は, 文字 n グラムモデルを用いて和歌の分析を行った最初期の研究です. 人文学におけるデータ解析の教科書としては, Python を用いた実例を示した “Humanities Data Analysis” [42]が 2021 年に出版されています. 社会科学におけるテキストデータの使用については, 5 章の文献案内をご覧ください.

∗65　https://rycolab.io/classes/info-theory-tutorial/
∗66　https://www.ism.ac.jp/~iba/a19.pdf