14章
自然言語のコーパスデータ

Peter Norvig

ビューティフルデータというとき、本書のほとんどの章ではボードレールのいう「美しいもの高尚なものはすべて理性と計算の結果である」という意味において美しいデータを扱っている。この章のデータはソローのいう「すべての人間が真に惹きつけられるのは自然体の言葉が持つ美しさである」という意味において美しい。我々が研究するデータは、ウェブページで公開されている1兆個にのぼる英単語であり、ありのままの言葉の極みである。スペルや文法の間違い、猫のおもしろ画像サイトLOLCATS、リックローリングなどの釣りリンクといったウェブが持つ猥雑さと共に、マーク・トウェイン、チャールズ・ディケンズ、ジェーン・オースティンやその他数え切れない著者の作品が並ぶ。

この1兆語のデータセットはGoogleのThorsten BrantsとAlex Franzによって2006年に公開され、Linguistic Data Consortium (http://tinyurl.com/ngrams) から入手可能だ。このデータセットは、元のテキストを1語の出現回数、2語、3語、4語、5語のシーケンス（語の連なり）の出現回数を数えることで集約している。例えば"the"は230億回（1兆語の2.2%）現れ、最も頻出する語となっている。"rebating"という語も、"fnuny"（明らかにfunnyの間違い）という語も、12750回（1%の100万分の1）現れる。3語のシーケンスでは、"Find all posts"が1300万回（0.001%）現れて"each of the"といい勝負になっているが、1億回現れる"All Rights Reserved"（0.01%）には敵わない。3語のシーケンスからいくつか抜き出してみよう。

```
outraged many African      63
outraged many Americans   203
outraged many Christians   56
outraged many Iraqis       58
outraged many Muslims      74
outraged many Pakistanis  124
outraged many Republicans  50
outraged many Turks       390
outraged many by           86
outraged many in          685
outraged many liberal      67
```

```
outraged many local        44
outraged many members      61
outraged many of          489
outraged many people      444
outraged many scientists   90
```

これを見ると、例えば（ウェブ上で、このデータが収集された時期に）トルコ人（Turks）は最も憤慨させられた（outraged）集団であり、共和党支持者（Republicans）とリベラル派（liberals）は時折憤慨させられるが、民主党支持者（Democrats）と保守派（conservatives）はリストに入っていないことがわかる。

なぜこのデータが美しいと言えるのだろうか。ただのありふれた日常ではないか。そう、それぞれのデータ単体ではありふれている。しかし億の単位で集約したものは美しい。それは英語という言語そのものにとどまらず、その言語を話す者が住む世界について多くのことを語ってくれる。語るに値する事柄を表現しているからこそ、このデータは美しいのである。

このデータを使って何ができるのかを見る前に、本章に現れる少しばかりの用語について話しておきたい。テキストを集めたものをコーパスと呼ぶ。我々はコーパスを、ワード（語）や句読点といったトークンのシーケンスとして扱う。他と識別可能な個別のトークンをタイプと呼ぶ。従って、テキスト "Run, Lola Run" は4つのトークン（カンマもトークンとして数える）を持つ一方で、タイプは3つしかない。すべてのタイプの集合をボキャブラリと呼ぶ。このGoogleのコーパスは1兆個のトークンと1300万個のタイプを持つ。英語の辞書にはおよそ100万語しか載っていないが、このコーパスには "www.njstatelib.org"、"+170.002"、"1.5GHz/512MB/60GB"、"Abrahamovich" といったタイプも含まれている。タイプのほとんどは出現回数が極度に少ない一方で、最頻出の10タイプだけで1/3近くのトークンを占め、最頻出1000タイプになると2/3以上をも占め、そして最頻出10万タイプで98%を占める。

トークン1つのシーケンスをユニグラム（unigram）、トークン2つのシーケンスをバイグラム（bigram）、そしてトークンn個のシーケンスをn-gramと呼ぶ。Pを確率（probability）としたとき、P(the) = .022はトークン "the" が出現する確率が.022（2.2%）であることを意味する。Wがトークンのシーケンスであるとき、W_3は3つめのトークンを意味し、$W_{1:3}$は1つめから3つめまでのトークンのシーケンスを意味する。$P(W_i = \text{the} \mid W_{i-1} = \text{of})$は "the" の前に "of" が現れる場合の条件付き確率（conditional probability）である。

このGoogleのコーパスについてもう少々説明しておく。200回未満しか現れないワードは未知（unknown）として<UNK>というシンボルで表される。40回未満しか現れないn-gramは除外されている。この方針により、文字の打ち間違いの影響を減らし、圧縮状態でほんの24GBで済ませることができている。最後に、コーパス中のそれぞれのセンテンスは特別なシンボル<S>と</S>で囲まれている。

さて、このデータを使って、いくつかのことを調べてみよう。

ワードセグメンテーション

中国語のテキスト「浮法像蝴蝶」について考える。これは「蝶のように浮かぶ」を意味する。5つの文字からなるが、英語のように文字間にスペースがない。従って中国語を読む時は、どこにワードの境界があるのかを決めるワードセグメンテーションを自分で行わなくてはならない。英文の場合はワード間にスペースがあるので、英語を読む時は普通はやらなくてもいい作業だ。しかしURLなどのテキストはスペー

スを持たないし、時には書き手が誤ってスペースを入れ忘れることもある。検索エンジンやワードプロセッサプログラムなどがこのような誤りを正すにはどうすればよいだろうか。

今度は英語のテキスト "choosespain.com" について考える。これは旅行先としてスペインを選んでもらおうと訪問者を誘うウェブサイトだが、誤って読めば "chooses pain (痛みを選ぶ)" と魅力に乏しい名前になる。人間は長年にわたる経験から正しい選択ができるが、その経験をコンピュータのアルゴリズムに組み込むというのは恐ろしく困難な仕事だろう。しかし我々は驚くほどうまく働く近道をとることができる。それぞれのフレーズをバイグラムテーブルから探すのである。"choose Spain" は3210回現れるのに対して、"chooses pain" は1回も現れない (1兆語のコーパス中に40回未満しか現れないことを意味する)。こうして、"choose Spain" が少なくとも80倍の確率で現れることから、これが正しいセグメンテーションであると安心して考えることができる。

次に "insufficientnumbers" というフレーズを解釈するという仕事を与えられたとしよう。大文字小文字を統一して足し合わせると、出現回数は以下のようになる。

```
insufficient numbers  20751
in sufficient numbers 32378
```

"in sufficient numbers (十分な数で)" の方が "insufficient numbers (数が不足して)" よりも50%多く出現しているが、これは確実な決め手とは言えない。決めたくても決められない。こちらかもしれないと言うことはできるが、自信はない。このような不明確な問題に対しては、我々は決定的な正しい答を計算する手段を持たず、どちらが正しい答であるかを決める完全なモデルを持たず、そして実のところ人間の専門家であっても完全なモデルを持たないばかりか、人間が出した答にさえ反論できる。それでも、不明確な問題を解くために確立された以下のような方法論がある。

1. 確率モデルを定義する

   ある分野において "choose Spain" を候補として優遇するためのすべての要素 (意味的、文法的、単語、社会的) を定義することはできないが、近似的な確率を与える単純化したモデルを定義することはできる。"choose Spain" のような短い候補に対しては、単にGoogleのコーパスデータ中のn-gramを参照して、その出現頻度を確率として用いればよい。長い候補に対しては、何らかの方法で短い部分から答を組み立てていかねばならない。過去に出会ったことのない単語の場合、その未知の単語の確率を見積もる必要がある。要は、対象とする言語におけるすべての文字列の確率分布として「言語モデル (language model)」を定義し、コーパスデータからモデルのパラメータを学習し、そのモデルを用いて各々の候補の確率を算出すればよい。

2. 候補を列挙する

   "insufficient numbers" と "in sufficient numbers" のどちらが意図した文であるか、わからないこともある。しかし両者とも区切り方として良い候補であることは確かだ。少なくとも "in suffi cient numb ers" や "hello world" よりは候補として適切だ。この場合、判断を保留して候補を列挙すればよい。できるならすべての候補を列挙するか、注意深く選び出したものを提示しよう。

3. 最もありえそうな候補を選ぶ
それぞれの候補に言語モデルを適用することにより確率を得て、最も確率の高いものを選ぶ。

数学的な方程式による表現を好むなら、こうなる。

最良候補 = argmax $c \in$ 候補 P(c)

プログラムコードの方が良いのであれば、こうなるだろう。

```
最良候補 = max(c in 候補, key=P)
```

前述の手法をセグメンテーションに適用してみよう。まず、スペースを除いた文字列を入力として取り、最も確からしいワードのリストを返す関数segmentを定義する。

```
>>> segment('choosespain')
['choose', 'spain']
```

最初のステップは確率的言語モデルだ。あるワードに着目するとしよう。そのワードに先行するワードを用い、それぞれのワードの出現確率を掛け合わせてワードシーケンスの確率を求める。数式で表現するとこうなる。

$$P(W_{1:n}) = \Pi_{k=1:n} P(W_k \mid W_{1:k-1})$$

これを正確に計算するに足るデータを我々は持っていないので、より狭い範囲のコンテキストを用いて近似計算する。最大5-gramまでのシーケンスデータがあることから、ついつい5-gramを用いて、(全先行ワードではなく)先行する4ワードを掛け合わせてnワードシーケンスの確率としてしまいたくなる。
しかし5-gramモデルには3つの問題がある。まず5-gramデータは30GB前後あるのでメモリに収まらない。そして大部分の5-gramカウント数は0であるため、「バックオフ」してより短いシーケンスから5-gramを見積もらねばならない。最後に、最大で4ワードに依存するため、候補の検索空間が大きくなってしまう。多少の努力で、これら3つの問題に対処することはできる。しかしその前に、より単純化した言語モデルでこの3つの問題すべてを一度に解決することを考えてみよう。それは、各ワードの出現確率が他のワードから独立している、ユニグラムモデルである。

$$P(W_{1:n}) = \Pi_{k=1:n} P(W_k)$$

'wheninrome'をセグメント化するとき、我々はwhen in romeのような候補を考え、P(when)×P(in)×P(rome)を計算する。この積が他の候補の積よりも大きければ、それが最善の解となる。
n文字の文字列は2^{n-1}通りの異なるセグメント化が可能だ(文字間にn-1個の狭間があり、それぞれがワード境界になり得る)。従って文字列'wheninthecourseofhumaneventsitbecomesnecessary'は35兆通

りのセグメント化が可能である。しかし読者のみなさんは、ものの数秒で正しいセグメント化ができることだろう。すべての候補を検討するというのはあり得ない。読者はおそらく "w"、"wh"、"whe" と見ていき、ワードとして不適格であるとしてそれらを捨てていった一方で、"when" はあり得そうなワードとして受け入れた。そして読者は残りの部分に移り、最もぴったりくるセグメンテーションを見つけ出した。つまりそれぞれのワードを独立したものと仮定して簡略化すれば、すべてのワードの組み合わせを考える必要がなくなる。

これがsegment関数の骨子となる。テキストを最初のワードと残りのテキストに分けるときにすべての可能な候補を考える（ワードの最大長はL=20文字というように任意に定めることができる）。可能な分割それぞれに対して、残りのテキストを分割する最良の区切り方を見つける。すべての可能な候補の中で、積P(1番目)×P(残り)が最も大きくなるものが最良の分割である。

ここで、最初のワードの選び方によって、最初のワードの確率、残りのテキストの最良分割における確率、全体の確率（最初のワードの確率と残りのテキストの最良分割確率の積）がどう変わるかを表にしてみよう。これを見ると、"when" で始まる分割は、2番目に良い分割より5万倍も良い分割であることがわかる。

1番目	P(1番目)	P(残り)	P(1番目)×P(残り)
w	$2 \cdot 10^{-4}$	$2 \cdot 10^{-33}$	$6 \cdot 10^{-37}$
wh	$5 \cdot 10^{-6}$	$6 \cdot 10^{-33}$	$3 \cdot 10^{-38}$
whe	$3 \cdot 10^{-7}$	$3 \cdot 10^{-32}$	$7 \cdot 10^{-39}$
when	$6 \cdot 10^{-4}$	$7 \cdot 10^{-29}$	$4 \cdot 10^{-32}$
wheni	$1 \cdot 10^{-16}$	$3 \cdot 10^{-30}$	$3 \cdot 10^{-46}$
whenin	$1 \cdot 10^{-17}$	$8 \cdot 10^{-27}$	$8 \cdot 10^{-44}$

segmentは数行のPythonで実装できる。

```
@memo
def segment(text):
    "テキストを最良分割した際のワードのリストを返す"
    if not text: return []
    candidates = ([first]+segment(rem) for first,rem in splits(text))
    return max(candidates, key=Pwords)

def splits(text, L=20):
    "すべての組み合わせの中からlen(1番目)<=Lである(1番目, 残り)のリストを返す"
    return [(text[:i+1], text[i+1:])
            for i in range(min(len(text), L))]

def Pwords(words):
    "ワードのシーケンスに対するナイーブベイズ確率"
```

```
    return product(Pw(w) for w in words)
```

これがプログラムのすべてである。ただし3カ所ほど省略している。productは数値リストの要素の積をとるユーティリティ関数、memoは関数呼び出しの結果をキャッシュして次回以降の再計算を不要にするデコレータ、Pwはユニグラムのカウントデータを用いてワードの確率を求める関数だ。

memo無しの場合、n文字のテキストに対してsegmentを呼び出すと、segmentが2^n回の再帰呼び出しを受ける。memo付きの場合、n回の呼び出しで済む。memoのおかげで非常に効率的な動的計画法(dynamic programming)アルゴリズムになる。n回の呼び出しそれぞれでO(L)個の分割を考え、それぞれの分割でO(n)個の確率を掛け合わせることで評価するので、アルゴリズム全体の計算量は$O(n^2L)$となる。

Pwはデータファイルからユニグラムのカウントを読み込む。コーパス中にワードが見つかった場合、見積もり確率はCount(word)/Nとなる。ここでNはコーパスのサイズである。実際には1300万のタイプを有する全ユニグラムを含むデータファイルではなく、vocab_commonという、(a)大文字小文字を区別しないようにするために"the"、"The"、"THE"などのカウントを一つのエントリ"the"にまとめ、(b)数字や記号を持たない、文字からなるワードのみを残し(つまり"+170.002"や"can't"は対象外)、(c)頻出100万ワードの中の上位1/3のみ含めたものを作成した。

ワードがコーパス中に存在しない場合を考えて、Pwにちょっとした工夫を盛り込んだ。1兆ワードを擁するコーパスでも時折発生するので、確率として0を返してはいけない。ではいくつにすべきか。コーパスのトークン数Nは、約1兆である。そしてvocab_commonの中で最も出現回数の少ないワードは12711回出現している。従って知らないワードは0と12710/Nの間の確率を持つはずだ。見知らぬワードすべてが同じ確率を持つわけではない。6文字のランダム文字列よりも20文字のランダム文字列の方がより起こり得ない。そこで確率分布を計算するためのクラスPdistを定義しよう。これにデータファイルから(key, count)のペアを読み込ませる。デフォルトではコーパスに存在しないワードの確率は1/Nとするが、Pdistのインスタンスそれぞれでカスタム関数を定義してデフォルト値をオーバーライドできる。非常に長いワードに高すぎる確率を割り当てるのを防ぐため、我々は(かなり適当ではあるが)確率10/Nから開始して、候補ワード中の1文字につき1/10を掛けて減らすことにした。そしてPwをPdistとして定義した。

```
class Pdist(dict):
    "データファイル中のカウント数から計算した確率分布"
    def __init__(self, data, N=None, missingfn=None):
        for key,count in data:
            self[key] = self.get(key, 0) + int(count)
        self.N = float(N or sum(self.itervalues()))
        self.missingfn = missingfn or (lambda k, N: 1./N)
    def __call__(self, key):
        if key in self: return self[key]/self.N
        else: return self.missingfn(key, self.N)
```

```
def datafile(name, sep='\t'):
    "ファイルからkey, valueの組を読み込む"
    for line in file(name):
        yield line.split(sep)

def avoid_long_words(word, N):
    "知らないワードの確率を計算する"
    return 10./(N * 10**len(word))

N = 1024908267229 ## コーパス中のトークン数

Pw = Pdist(datafile('vocab_common'), N, avoid_long_words))
```

Pw[w]はワードwのカウントそのものであるのに対して、Pw(w)は確率であることに注意してもらいたい。この章に載せているすべてのプログラムはhttp://norvig.com/ngramsで配布している。

さて、このモデルはどの程度よいセグメント化をするのだろうか。いくつか例を挙げてみる。

```
>>> segment('choosespain')
['choose', 'spain']
>>> segment('thisisatest')
['this', 'is', 'a', 'test']
>>> segment('wheninthecourseofhumaneventsitbecomesnecessary')
['when', 'in', 'the', 'course', 'of', 'human', 'events', 'it', 'becomes',
'necessary']
>>> segment('whorepresents')
['who', 'represents']
>>> segment('expertsexchange')
['experts', 'exchange']
>>> segment('speedofart')
['speed', 'of', 'art']
>>> segment('nowisthetimeforallgood')
['now', 'is', 'the', 'time', 'for', 'all', 'good']
>>> segment('itisatruthuniversallyacknowledged')
['it', 'is', 'a', 'truth', 'universally', 'acknowledged']
>>> segment('itwasabrightcolddayinaprilandtheclockswerestrikingthirteen')
['it', 'was', 'a', 'bright', 'cold', 'day', 'in', 'april', 'and', 'the', 'clocks', 'were',
'striking', 'thirteen']
>>> segment('itwasthebestoftimesitwastheworstoftimesitwastheageofwisdomitwastheage
offoolishness')
```

```
['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times',
'it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of', 'foolishness']
>>> segment('asgregorsamsaawokeonemorningfromuneasydreamshefoundhimselftransformed
inhisbedintoagiganticinsect')
['as', 'gregor', 'samsa', 'awoke', 'one', 'morning', 'from', 'uneasy', 'dreams', 'he',
'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'gigantic', 'insect']
>>> segment('inaholeinthegroundtherelivedahobbitnotanastydirtywetholefilledwiththe
endsofwormsandanoozysmellnoryetadrybaresandyholewithnothinginittositdownonortoeat
itwasahobbitholeandthatmeanscomfort')
['in', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'hobbit', 'not',
'a', 'nasty', 'dirty', 'wet', 'hole', 'filled', 'with', 'the', 'ends', 'of', 'worms',
'and', 'an', 'oozy', 'smell', 'nor', 'yet', 'a', 'dry', 'bare', 'sandy', 'hole', 'with',
'nothing', 'in', 'it', 'to', 'sitdown', 'on', 'or', 'to', 'eat', 'it', 'was', 'a',
'hobbit', 'hole', 'and', 'that', 'means', 'comfort']
>>> segment('faroutintheunchartedbackwatersoftheunfashionableendofthewesternspiral
armofthegalaxyliesasmallunregardedyellowsun')
['far', 'out', 'in', 'the', 'uncharted', 'backwaters', 'of', 'the', 'unfashionable',
'end', 'of', 'the', 'western', 'spiral', 'arm', 'of', 'the', 'galaxy', 'lies', 'a',
'small', 'un', 'regarded', 'yellow', 'sun']
```

"Samsa" や "oozy" のように見慣れない単語が正しくセグメント化されているのを見て、読者は満足していることだろう。しかし驚くなかれ、1兆ワードのコーパスの中に "Samsa" は42000回出現し、"oozy" は13000回出現している。全体として結果は良好であるように見えるが、間違いが2つある。'un','regarded'は1ワードになるべきだし、'sitdown'は2ワードになるべきだ。それでもワードの正確性は157/159 = 98.7%にもなる。悪くない。

1つめの間違い "unregarded" は、100万ワードの1/3のボキャブラリ中に現れない（全ボキャブラリにまで広げると1005493位で7557回現れている）。このワードをボキャブラリに含めておけば、正しくセグメント化される。

```
>>> Pw['unregarded'] = 7557
>>> segment('faroutintheunchartedbackwatersoftheunfashionableendofthewesternspiral
armofthegalaxyliesasmallunregardedyellowsun')
['far', 'out', 'in', 'the', 'uncharted', 'backwaters', 'of', 'the', 'unfashionable',
'end', 'of', 'the', 'western', 'spiral', 'arm', 'of', 'the', 'galaxy', 'lies', 'a',
'small', 'unregarded', 'yellow', 'sun']
```

これで問題が解決したことにはならない。この1ワードのみならず、このワードより上位のワードすべてを組み込んで、全テストケースを実行し、ワードの追加が他の結果に悪影響を出さないことを確認しなくてはならない。

2つめの間違いは、"sit" も "down" も頻出ワード（それぞれ0.003%と0.04%）であるにもかかわらず、この2つの確率を掛け合わせると "sitdown" の確率よりも若干低くなってしまうために発生している。しかし、バイグラムのカウントを見ると、2ワードの "sit down" の方が100倍も高確率で現れる。この問題をバイグラムを用いたモデリングで解決するよう努力することもできる。つまり、それぞれのワードの確率を、1つ前のワードを考慮して以下のように算出する。

$$P(W_{1:n}) = \Pi_{k=1:n} P(W_k | W_{k-1})$$

当然ながら完全なバイグラムテーブルはメモリに乗らない。10万回以上現れたバイグラムのみ保持するようにすれば、25万エントリを若干超えるくらいになり、メモリに収まる。すると P(down | sit) は Count(sit down)/Count(sit) と計算できる。あるバイグラムがテーブル中に見あたらない場合は、ユニグラムの利用へと縮退動作させる。直前のワードに続く、あるワードの条件付き確率cPwは以下のように定義できる。

```
def cPw(word, prev):
    "条件付き確率 P(ワード | 直前のワード)"
    try:
        return P2w[prev + ' ' + word]/float(Pw[prev])
    except KeyError:
        return Pw(word)

P2w = Pdist(datafile('count2w'), N)
```

直前のワード付きですべてのワードの総和を取ると1よりも大きくなり得るのでcPwは確率分布ではない、と細かい人は言うかもしれない。この手法は、ステュービッド・バックオフ（stupid backoff：愚かな譲歩）という専門用語で呼ばれているのだが、現実によく機能するので我々は気にしていない。さてここで、直前ワードが "to" の場合に、それに続く "sitdown" と "sit down" を比較してみる。

```
>>> cPw('sit', 'to')*cPw('down', 'sit') / cPw('sitdown', 'to')
1698.0002330199263
```

"sit down" は "sitdown" よりも1698倍も確からしいことがわかる。これは "sit down" が頻出バイグラムであることと、"to sit" が頻出する一方で "to sitdown" はあまり現れないことによる。

これで完璧であるように見える。バイグラムモデルを用いてsegmentの新バージョンを実装してみよう。その際、次の2つの課題に対処する。

1. segmentがnワードのシーケンスに新ワードを1つ追加するとき、n+1ワードのすべての確率の積をとろうとPwordsが呼ばれる。しかしsegmentはnワードの確率の積をすでに計算済みだ。計算済みの確率を記憶しておくようにすれば、積を1回とるだけで済むので効率的だ。

2. 算術アンダーフローが起こりうるという問題がある。"blah" が61回連続するワードのシーケンスをPwordsに与えると$5.2*10^{-321}$が得られる。これにさらに1つ "blah" を付け足すと、0.0になってしまう。表現可能な中で最も小さな正の浮動小数点数は$4.9*10^{-324}$である。これより小さなものはすべて0.0に丸められてしまう。アンダーフローを防ぐための最も簡単な方法は、数そのものの積をとるのではなく、対数の和をとることである。

ここでsegment2を定義する。segmentとは3つの点で異なる。まず、ユニグラムモデルPwではなく、条件付きバイグラムモデルcPwを利用している。さらに関数のシグネチャも異なる。1つの引数textに加えて、segment2は直前のワードも渡す。センテンスの開始時の直前のワードは、センテンス開始印<S>である。戻り値は単なるワードのリストではなく、セグメンテーションの確率とワードのリストの組にした。確率を返すのは、確率を（memoによって）保存して再計算を不要にするためである。これで1つめの問題である非効率性が解決する。関数combineは4つの入力を持つ。最初のワード、残りのワード、それらの確率である。最初のワードを残りのワードに結合し、確率の積をとる。ただし2つめの問題に対処するために、もうひとつの違いを導入する。生の確率を掛けるのではなく、確率の対数を足すのである。

segment2のコードはこうなる。

```
from math import log10

@memo
def segment2(text, prev='<S>'):
    "最良分割となる(log P(ワードリスト), ワードリスト)を返す"
    if not text: return 0.0, []
    candidates = [combine(log10(cPw(first, prev)), first, segment2(rem, first))
                  for first,rem in splits(text)]
    return max(candidates)

def combine(Pfirst, first, (Prem, rem)):
    "最初のワードと残りのワードリストをひとつの(確率, ワードリスト)の組へ結合する"
    return Pfirst+Prem, [first]+rem
```

segment2はO(nL)回の再帰呼び出しをする。それぞれの呼び出しにおいてO(L)個の分割を検討するため、アルゴリズム全体の計算量は$O(nL^2)$となる。結果として、これはビタビ（Viterbi）アルゴリズムになっている。memoは暗黙的にビタビテーブルを作成する。

segment2は "sit down" の例文を正しくセグメント化し、最初のバージョンで正しい結果が得られていた他のすべての例文においても正しい結果を返す。しかしどちらのバージョンも "unregarded" の例文を正しくセグメント化することはできない。

これを改善することはできるだろうか。おそらくできるだろう。未知のワードに対するより正確なモデルを作成することもできる。より多くのデータを取り入れて、ユニグラムあるいはバイグラムのエントリをより多く保持することもできる。トライグラムのデータを使うこともできるかもしれない。

暗号

我々の次の課題は暗号で書かれたメッセージの解読だ。これから、文字一つひとつを別の文字に置き換えていく換字式暗号（かえじしきあんごう：substitution cipher）を見ていく。どの文字をどの文字へ置き換えるのかという指示を鍵（key）と呼ぶ。ここでは26文字の文字列で表す。1文字目がaを置き換え、2文字目がbを置き換える、といった具合である。メッセージを換字暗号鍵でエンコードする関数を以下に示す（処理の大部分をPythonのライブラリ関数maketransとtranslateが受け持っている）。

```
def encode(msg, key):
    "換字式暗号でメッセージをエンコードする"
    return msg.translate(string.maketrans(ul(alphabet), ul(key)))

def ul(text): return text.upper() + text.lower()

alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

すべての暗号の中でも最も単純なものはシフト暗号（shift cipher）であろう。これは換字式暗号の一種で、メッセージ中のそれぞれの文字をアルファベットのn文字後ろの文字で置き換えるものだ。n=1の場合、aはbに、bはcに置き換えられ、そしてzはaに置き換えられる。シフト暗号はシーザー暗号（Caesar cipher）とも呼ばれる。紀元前50年当時は最先端技術であった。関数shiftはシフト暗号によるエンコードを行う。

```
def shift(msg, n=13):
    "シフト暗号（シーザー暗号）でメッセージをエンコードする"
    return encode(msg, alphabet[n:]+alphabet[:n])
```

この関数は以下のように用いる。

```
>>> shift('Listen, do you want to know a secret?')
'Yvfgra, qb lbh jnag gb xabj n frperg?'

>>> shift('HAL 9000 xyz', 1)
'IBM 9000 yza'
```

鍵がわからない時にメッセージをデコードするには、セグメント化と同じ手法に則ればよい。モデルを定義して（ここではユニグラムのワード出現確率を用いる）、候補を列挙し、最もあり得そうなものを選ぶ。全部で26候補しかないので、すべてを試すことができる。

これをlogPwordsとして実装する。Pwordsに似ているが、確率の対数を返し、入力として複数のワードからなる文字列とワードのリストのいずれもとることができる。

```
def logPwords(words):
    "文字列またはワードシーケンスのナイーブベイズ確率"
    if isinstance(words, str): words = allwords(words)
    return sum(log10(Pw(w)) for w in words)

def allwords(text):
    "アルファベットのみからなるワードのリストを小文字にして返す"
    return re.findall('[a-z]+', text.lower())
```

これで、すべての候補を列挙して最もあり得そうなものを取り出すことでデコードできるようになった。

```
def decode_shift(msg):
    "シフト暗号器でエンコードされたメッセージをデコードし、最もあり得そうなものを見つける"
    candidates = [shift(msg, n) for n in range(len(alphabet))]
    return max(candidates, key=logPwords)
```

次のようにして動作確認する。

```
>>> decode_shift('Yvfgra, qb lbh jnag gb xabj n frperg?')
'Listen, do you want to know a secret?'
```

なんということはない。26個の候補とその対数確率を見れば理由がわかる。

```
Yvfgra, qb lbh jnag gb xabj n frperg? -84
Zwghsb, rc mci kobh hc ybck o gsqfsh? -83
Axhitc, sd ndj lpci id zcdl p htrgti? -83
Byijud, te oek mqdj je adem q iushuj? -77
Czjkve, uf pfl nrek kf befn r jvtivk? -85
Daklwf, vg qgm osfl lg cfgo s kwujwl? -91
Eblmxg, wh rhn ptgm mh dghp t lxvkxm? -84
Fcmnyh, xi sio quhn ni ehiq u mywlyn? -84
Gdnozi, yj tjp rvio oj fijr v nzxmzo? -86
Heopaj, zk ukq swjp pk gjks w oaynap? -93
Ifpqbk, al vlr txkq ql hklt x pbzobq? -84
Jgqrcl, bm wms uylr rm ilmu y qcapcr? -76
Khrsdm, cn xnt vzms sn jmnv z rdbqds? -92
Listen, do you want to know a secret? -25
Mjtufo, ep zpv xbou up lopx b tfdsfu? -89
```

```
Nkuvgp, fq aqw ycpv vq mpqy c ugetgv? -87
Olvwhq, gr brx zdqw wr nqrz d vhfuhw? -85
Pmwxir, hs csy aerx xs orsa e wigvix? -77
Qnxyjs, it dtz bfsy yt pstb f xjhwjy? -83
Royzkt, ju eua cgtz zu qtuc g ykixkz? -85
Spzalu, kv fvb dhua av ruvd h zljyla? -85
Tqabmv, lw gwc eivb bw svwe i amkzmb? -84
Urbcnw, mx hxd fjwc cx twxf j bnlanc? -92
Vscdox, ny iye gkxd dy uxyg k combod? -84
Wtdepy, oz jzf hlye ez vyzh l dpncpe? -91
Xuefqz, pa kag imzf fa wzai m eqodqf? -83
```

このリストを見れば、まさに1つの行だけが英語的であることがわかる。Pwordsもまた我々の直感を擁護して、この行の対数確率は-25（つまり10^{-25}）であり、他のどの候補より10^{50}倍も確からしいことがわかる。

暗号作成者は、記号やワード間の空白を取り除き、大文字小文字の区別をなくして暗号解読者をより困らせることもできる。このようにすると、短い単語I、a、theなどから手がかりを得られなくなるし、アポストロフィーに続く文字はsやtであると推測することもできなくなる。ここで、非文字を取り除き、すべてを小文字に変換してからシフト暗号を適用するshift2を作成しよう。

```
def shift2(msg, n=13):
    "シフト(シーザー)暗号でエンコードして[a-z]の文字のみを生成する"
    return shift(just_letters(msg), n)

def just_letters(text):
    "テキストを小文字化して[a-z]以外のすべての文字を削除する"
    return re.sub('[^a-z]', '', text.lower())
```

暗号は以下のようにして破る。それぞれの候補を列挙してセグメント化し、最も高い確率を持つものを選ぶ。

```
def decode_shift2(msg):
    "シフト暗号でエンコードされた、空白を含まないメッセージをデコードする"
    candidates = [segment2(shift(msg, n)) for n in range(len(alphabet))]
    p, words = max(candidates)
    return ' '.join(words)
```

動作を見てみよう。

```
>>> shift2('Listen, do you want to know a secret?')
'yvfgraqblbhjnaggbxabjnfrperg'

>>> decode_shift2('yvfgraqblbhjnaggbxabjnfrperg')
'listen do you want to know a secret'

>>> decode_shift2(shift2('Rosebud'))
'rosebud'

>>> decode_shift2(shift2("Is it safe?"))
'is it safe'

>>> decode_shift2(shift2("What's the frequency, Kenneth?"))
'whats the frequency kenneth'

>>> msg = 'General Kenobi: Years ago, you served my father in the Clone
Wars; now he begs you to help him in his struggle against the Empire.'

>>> decode_shift2(shift2(msg))
'general kenobi years ago you served my father in the clone wars now he
begs you to help him in his struggle against the empire'
```

これはまだ易しいだろう。次は、任意の文字を別の任意の文字へ置き換える一般的な換字式暗号を見てみよう。鍵が26種類から26!種類（約4×10^{26}）に増えるので、もはや候補を列挙することはできない。サイモン・シンの「暗号解読」によると、暗号破りには5つの戦略（我々はこれにもう1つ付け加える）がある。

1. 文字ユニグラムの頻度。メッセージ中の頻出文字と英語の頻出文字（eのような）、そして最も出現しない文字（zのような）を対応させる。
2. 2連続文字の分析。暗号化されたメッセージ中で同じ文字が2文字連続していれば、デコードされたメッセージ中でも連続している。最も少ない、そして最も多い2連続文字を考える。
3. the、and、ofなどの頻出ワードを探す。1文字ワードのほとんどはaとIだけだ。
4. 可能であれば、扱うメッセージの種類に応じて頻度テーブルを分けて用意する。例えば軍事関係のメッセージには軍事用語が使われている。
5. ワードやフレーズを推測する。例えばメッセージに "your faithful servant（昔の手紙の結びで使われた言葉）" が含まれていると思われるのであれば、それを試してみよ。
6. ワードのパターンを用いる。例えば暗号化されたワードが "abbccddedf" であれば、これは "bookkeeper" である可能性が高い。コーパス中にこれと同じパターンを持つワードは存在しない。

ワード間に空白を挟まないメッセージの場合、戦略3から6は適用できない。戦略1と2はそれぞれ26

の可能性しか持たないので、コンピュータプログラムではなく、メモリも計算能力も限られた人間の分析者に向けたものであろう。戦略4と5は、一般的な解読者ではなく、特別な目的を持つ解読者のためのものである。最後の戦略は自前で導き出したように見えるが、我々は以下の方法論を知っている。

I. 確率モデルを定義する

シフト暗号のときのように、テキストをセグメント化してワードの確率を計算するという同じ方法で候補を評価できる。しかしこの方法論の2番目のステップを考えると、最初のいくつか（というより最初の数千）の候補はまるであり得ないものになるだろう。探索の開始時には似たワードが全く見つからないので、セグメント化を試みてもはかばかしい進展はない。しかし、ある一行の中で（偶然に）意味の通じる数文字がデコードされることがある。ここで、言語モデルとしてワードn-gramではなく、文字n-gramを使ってみよう。文字バイグラムを使えばよいだろうか。あるいは3-gram、5-gramを使えばよいだろうか。私はここで、頻出する短いワードを表現できる3-gramを採用した（戦略3）。ワードバイグラムのデータファイル中の文字3-gram（空白と記号を取り除いたもの）を数えて、データファイルcount_3lを作成した。ワード境界をまたぐ文字トライグラムを考慮する必要があるため、単にボキャブラリファイルを見るだけでは済まない。全部で$26^3 = 17576$種のトライグラムが現れる。上位10個と下位10個を見てみよう。

```
the  2.763%          fzq  0.0000004%
ing  1.471%          jvq  0.0000004%
and  1.462%          jnq  0.0000004%
ion  1.343%          zqh  0.0000004%
tio  1.101%          jqx  0.0000003%
ent  1.074%          jwq  0.0000003%
for  0.884%          jqy  0.0000003%
ati  0.852%          zqy  0.0000003%
ter  0.728%          jzq  0.0000002%
ate  0.672%          zgq  0.0000002%
```

文字トライグラムの確率は以下のように算出する。

```
def logP3letters(text):
    "文字3-gramモデルを利用したテキストの対数確率"
    return sum(log10(P3l(g)) for g in ngrams(text, 3))

P3l = Pdist(datafile('count_3l'))
P2l = Pdist(datafile('count_2l')) ## We'll need it later
```

II.候補を列挙する

4×10^{26}個すべての可能な鍵を検討することはできないし、セグメント化のときのように体系的に不

適切な候補を消していく方法があるようにも思えない。このような場合には、山登り法（hill climbing）に代表される局所探索法（local search）を用いる。例えば一番高い所に上りたいときに地図がないとき、山登り法では任意の場所xから出発して近傍へ歩を進める。その場所がより高いところであれば山登りを続ける。高くなければxの別の近傍を試す。当然ながら、地球上の任意の場所から高みを目指して歩き始めても、おそらくエベレストの頂上にはたどり着かない。普通は近所の小さな丘の上か、平地をうろうろしてどこへも行けないかだろう。そこで我々は、山登り法にランダム再出発（random restart）を付け足して、ある歩数以上歩いたら別のランダムな場所から再出発するようにした。

一般的な山登り法アルゴリズムを以下に示す。出発地点x、最適解を見つけようとしている関数f、ある地点の近傍を返す関数neighbors、そして最大歩数を引数にとる（変数debuggingがtrueの場合、最適解となるxとそのスコアを出力する）。

```
def hillclimb(x, f, neighbors, steps=10000):
    "neighbors(x) を調べて f(x) を最大化する x を探す"
    fx = f(x)
    neighborhood = iter(neighbors(x))
    for i in range(steps):
        x2 = neighborhood.next()
        fx2 = f(x2)
        if fx2 >= fx:
            x, fx = x2, fx2
            neighborhood = iter(neighbors(x))
    if debugging: print 'hillclimb:', x, int(fx)
    return x

debugging = False
```

hillclimbをデコードに用いるためにパラメータを指定する必要がある。我々が探索するその地形とは、平文テキストメッセージである。最大化しようとするのは文字トライグラムだから、fとしてlogP3lettersを与える。まずランダムな鍵で復号化されたメッセージをxとして与えて処理を開始する。ランダム再出発を実行し、それぞれのランダム再出発から候補を集める際にはlogP3lettersではなくsegment2を最大化するものを選ぶ。

```
def decode_subst(msg, steps=4000, restarts=20):
    "ランダム再出発山登り法により換字式暗号を復号化する"
    msg = cat(allwords(msg))
    candidates = [hillclimb(encode(msg, key=cat(shuffled(alphabet))),
                            logP3letters, neighboring_msgs, steps)
                  for _ in range(restarts)]
```

```
        p, words = max(segment2(c) for c in candidates)
        return ' '.join(words)

    def shuffled(seq):
        "入力シーケンスをランダムに並べ替えたコピーを返す"
        seq = list(seq)
        random.shuffle(seq)
        return seq

    cat = ''.join
```

さて、次に試すメッセージを復号化するneighboring_msgsを定義する必要がある。まず最初に、あり得なさそうな文字バイグラムの修正を試みる。例えば最も出現頻度の低いバイグラム "jq" の出現確率0.0001%は、最頻出バイグラムinとthの5万分の1である。そこで、msgの中に "jq" を見つけた場合は、"j" を別の文字に置き換え、"q" も別の文字に置き換えようと試みる。この置き換えによってより頻度の高いバイグラムが生成された場合、この置き換えによって作られたメッセージを出力する。最も起こり得ないバイグラム上位20個を置換するという仕事を終えると、次はランダムに置き換える。

```
def neighboring_msgs(msg):
    "よりよい鍵を求めて近傍の鍵を生成する"
    def swap(a,b): return msg.translate(string.maketrans(a+b, b+a))
    for bigram in heapq.nsmallest(20, set(ngrams(msg, 2)), P2l):
        b1,b2 = bigram
        for c in alphabet:
            if b1==b2:
                if P2l(c+c) > P2l(bigram): yield swap(c,b1)
            else:
                if P2l(c+b2) > P2l(bigram): yield swap(c,b1)
                if P2l(b1+c) > P2l(bigram): yield swap(c,b2)
    while True:
        yield swap(random.choice(alphabet), random.choice(alphabet))
```

うまく動くかどうか、確かめてみよう。Robert Raynard著のSecret Code Breaker (暗号解読者——Smith and Daniel刊) からいくつかの暗号を拝借する。まずは小手調べのメッセージ。

```
>>> msg = 'DSDRO XFIJV DIYSB ANQAL TAIMX VBDMB GASSA QRTRT CGGXJ MMTQC IPJSB
AQPDR SDIMS DUAMB CQCMS AQDRS DMRJN SBAGC IYTCY ASBCS MQXKS CICGX RSRCQ ACOGA
SJPAS
AQHDI ASBAK GCDIS AWSJN CMDKB AQHAR RCYAE'
```

```
>>> decode_subst(msg)
'it is by knowing the frequency which letters usually occur and other
distinctive characteristics of the language that crypt analysts are able to
determine the
plain text of a cipher message j'
```

"crypt analysts" が1つのワードであるべきだという点を除いて、これは正しい答えだ（Pwにはないが1300万語のワードボキャブラリの中にはある）。なお、最後の文字（暗号テキストの "E"）は、すべてを5文字のブロックにして見た目を統一するために加えられている。

次は第一次世界大戦中にドイツのスパイ Baron August Schlugaが送信した実際のメッセージだ。

```
>>> msg = 'NKDIF SERLJ MIBFK FKDLV NQIBR HLCJU KFTFL KSTEN YQNDQ NTTEB TTENM
QLJFS NOSUM MLQTL CTENC QNKRE BTTBR HKLQT ELCBQ QBSFS KLTML SSFAI NLKBR RLUKT
LCJUK
FTFLK FKSUC CFRFN KRYXB'

>>> decode_subst(msg)
'english complaining over lack of munitions they regret that the promised
support of the french attack north of arras is not possible on account of
munition insufficiency wa'
```

次は1992年にKBGから元CIA職員Aldrich Amesに送られたメッセージだ。彼は1994年にスパイ容疑で有罪判決を受けている。

```
>>> msg = 'CNLGV QVELH WTTAI LEHOT WEQVP CEBTQ FJNPP EDMFM LFCYF SQFSP NDHQF
OEUTN PPTPP CTDQN IFSQD TWHTN HHLFJ OLFSD HQFED HEGNQ TWVNQ HTNHH LFJWE BBITS
PTHDT
XQQFO EUTYF SLFJE DEFDN IFSQG NLNGN PCTTQ EDOED FGQFI TLXNI'

>>> decode_subst(msg)
'march third week bridge with smile to pass info from you to us and to give
assessment about new dead drop ground to indicate what dead drop will be used
next to give your opinion about caracas meeting in october xab'
```

1943年にドイツのUボート指揮官から発せられたこのメッセージは傍受、解読され、同盟船の護送を助けた。

```
msg = 'WLJIU JYBRK PWFPF IJQSK PWRSS WEPTM MJRBS BJIRA BASPP IHBGP RWMWQ SOPSV
```

```
PPIMJ BISUF WIFOT HWBIS WBIQW FBJRB GPILP PXLPM SAJQQ PMJQS RJASW LSBLW GBHMJ
QSWIL PXWOL'

>>> decode_subst(msg)
'a cony ov is headed northeast take up positions fifteen miles apart between
point yd and bu maintain radio silence except for reports of tactical
importance x abc'
```

この答は "y" と "v" を混同している。人間が分析している場合、"cony ov" は "convoy" で、"point yd" は "point vd" であるはずだとわかる。しかし正解テキストの文字トライグラムの確率は上に示したものよりも低いため、このプログラムはそういった可能性を考えない。局所最大値にとらわれない、より優れたスコアリング関数を作り出すことによって、この問題に対処できるだろう。あるいは2段階目の山登り探索を追加して、最初の探索で見つかった候補をsegment2をスコアリング関数にして軽く探索することもできる。これらの工夫は読者にお任せしよう。

スペル訂正

最後に取り上げるのはスペル訂正だ。入力されたワードwから、どのワードcを入力しようとしたのかを見つけ出す。例えばwが "acomodation" である場合、cは "accommodation" になるだろう（wが "the" ならばcもまた "the" になる）。

標準的な方法論に則り、P(c | w)を最大化するcを選ぼう。しかしこの確率は一筋縄では定義できない。w= "thew" である場合について考える。候補cの一つとして "the" が考えられる。これは最頻出ワードであり、キーを打った人が "e" のキーから指を滑らせて "w" も打ったのだろうと想像できる。もう一つの候補として "thaw" がある。頻出ワードの一つ（とはいえ "the" より3万倍も頻度が低い）であるし、母音を他の母音に置き換えてしまうという間違いはよくある。他の候補として "thew（筋肉を意味する古語）" そのもの、"threw"、名字 "Thwe" が考えられる。どれを選ぶべきだろうか。これは2つの要素が合わさっているように見える。cそのものの確からしさと、wがcと打ち間違えられる、あるいは誤発音される、あるいは何らかのミススペルになる確率だ。これらの要素を何とかして統合しなければならないと考える人もいるだろうが、ベイズの定理という数学的な式がある。最良候補を見つけるために、これらをどのように統合すれば良いのかを正確に語ってくれる。

$$\text{argmax}_c\, P(c \mid w) = \text{argmax}_c\, P(w \mid c)\, P(c)$$

ここでP(c)はcが目的のワードである確率で、言語モデル（language model）と呼ばれる。P(w | c)は、書き手がcと書きたいにもかかわらずwと書いてしまう確率で、誤りモデル（error model）やノイジーチャンネルモデル（noisy channel model）と呼ばれる（理想的な書き手がcと書こうとしたときに、回線のノイズや妨害によってcがwに変わってしまうというのがその由来である）。しかし残念ながら、このモデルを手持ちのコーパスデータから簡単に構築する方法はない。コーパスには、どのワードがどのワードのミススペルであるかという情報がないのだ。

ミススペルのリストを用いることによりこの問題を解決できる。Roger Mittonは約4万のc,wの組を

http://www.dcs.bbk.ac.uk/~ROGER/corpora.htmlで公開している。しかしこのデータの中にP(w=thew | c=thaw)があるとは思えない。たった4万のデータでは、この組そのものが見つかる可能性は小さい。データが疎な場合、一般化する必要がある。この場合、同じ文字"th"と"w"を無視してP(w=e | c=a)を求める、つまり正しい文字が"a"である場合に"e"と打ってしまう確率を求めることで一般化できる。これはミススペルデータの中でも頻出する誤りで、"consistent/consistant"や"inseparable/inseperable"などの混同にも見られるものだ。

以下の表にはw=thewである場合の、5つの候補cを載せた。ひとつはthewそのもので、他の4つは1回の編集操作により生まれるものである。(1)"thew"中の文字"w"を「削除」して"the"とする。(2)"r"を「挿入」して"threw"とする。この2種類の編集は直前の文字に影響される。(3)前述のように"e"を"a"に置き換える。(4)隣接した2文字を入れ替えて"ew"を"we"とする。これらの単一編集操作を編集距離(edit distance)が1であるという。2回の単一編集操作が必要な候補は編集距離2となる。この表にはw、c、編集w | c、確率P(w | c)、確率P(c)、そして確率の積を読みやすくするために大きな数を掛けて載せた。

w	c	w | c	P(w | c)	P(c)	10^9P(w | c) P(c)
thew	the	ew | e	0.000007	0.02	144.
thew	thew		0.95	0.00000009	90.
thew	thaw	e | a	0.001	0.0000007	0.7
thew	threw	h | hr	0.000008	0.000004	0.03
thew	thwe	ew | we	0.000003	0.00000004	0.0001

この表を見ると"the"が最もあり得る訂正であることがわかる。P(c)はPwで計算できる。P(w|c)の計算には新しい関数Peditが必要だ。この関数は編集の確率をミススペルのコーパスから見積もる。例えばPedit('ew|e')は0.000007となる。より複雑な編集は単一の編集の結合として定義される。例えば"hallow"から"hello"を得る時、a|eとow|oを結合してn|e+ow|oと表記する。あるいはow|o+a|eと表記してもよい。常にではないが、この場合は同じ意味になる。複雑な編集の確率は、その部分部分の編集の積となる。

ここで1つ問題がある。編集なしの場合のPedit('')の確率はいくつになるのだろうか。つまり、cというワードを打ちたい場合に、書き手が打ち間違えずにcそのものを打つ可能性はどの程度あるのだろうか。これは入力者の能力にもよるし、何らかの校正がなされたかどうかにもよる。かなりいいかげんだが、私は20ワードに1回、スペルミスをすると仮定した。ちなみに50ワードに1回打ち間違えると仮定すると、w="thew"に対するP(w|c)は0.95ではなく0.98となり、"thew"が最もあり得そうな答えとなる。

最後にコードを見てみよう。2つの最上位関数がある。correctは1つのワードに対する最もあり得そうな訂正を返し、correctionsはテキスト中のそれぞれのワードに、周りの文字を固定したままcorrectを適用する。すべての可能な編集が候補となるが、スコアP(w|c) P(c)を最大にするものを採用する。

```
def corrections(text):
    "テキスト中のすべてのワードをスペル訂正する"
    return re.sub('[a-zA-Z]+', lambda m: correct(m.group(0)), text)
```