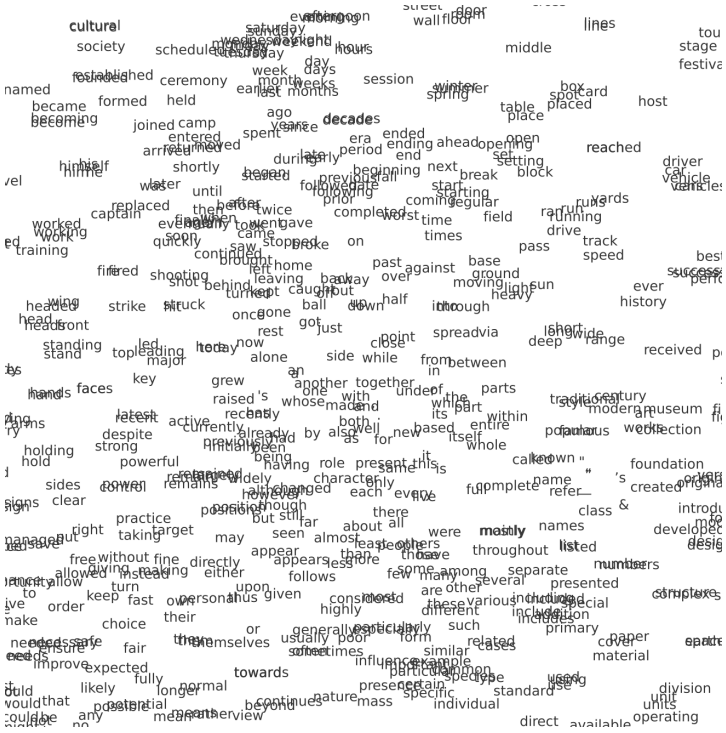
統計的テキストモデル

持橋 大地
統計数理研究所 統計基盤数理研究系
daichi@ism.ac.jp

2024 年 11 月 23 日

$Id: textmodel.tex,v 1.21 2024/11/19 22:14:09 daichi Exp $



はじめに

本書は, テキストの統計的なモデル化について解説した本です. ここでいうテキストとは, Web ページやメールのような文書はもちろん, 小説や新聞, 法案, アンケートへの自由回答など, さまざまな範囲を対象にしています. よって本書の読者としては, テキストを扱う必要のある理系の方だけでなく, 人文科学および社会科学系の方々, および言語に興味のある一般の方々を想定しています.

Web が現れて以降, 電子化されたテキストを扱う自然言語処理は驚異的に進展しました. しかし, 自然言語処理の教科書や情報源は, 文書のカテゴリや単語の品詞のように, 人の与えた正解ラベルをテキストから予測する**教師あり学習**を扱っている場合がほとんどで, 現実に現れる, そうした正解ラベルのないテキストを統計的にどうモデル化して扱うか, という**教師なし学習**について体系的にまとめられた成書は, ほとんど存在しないのが現状です. 「テキストマイニング」はこれに近い分野ですが, 多くの場合, 表面的なパッケージの使い方に終わっていたり, 離散的な言葉の頻度に従来の多変量解析を無理に当てはめていたりといった問題がありました. テキストのような離散データを取り扱うには, より適切な統計モデルが存在します. またその際に, 内部でどのような数学的なモデル化と計算が行われているのかがわからなければ, 目の前の問題に適用するためには, どこにどう手を入れたらよいのかを知ることはできません.

そこで本書では, テキストの統計的なモデル化について一から説明し, **ブラックボックスに頼らなくても, さまざまな分析を自分で自由に行えるようになる**ことを目的としています. 本書で説明するようなテキストの高度な統計モデルにはパッケージがないことも多く, また, パッケージでブラックボックス化して扱うことは適切とはいえません. あえて「背景知識」のような無味乾燥な章は作らず, 必要になる数学的知識はすべてその場で, 実例を交じえつつ説明してい

ますので, 本書を読むには高校レベルの数学を理解していれば充分で, あらかじめ機械学習の教科書を読んでおく必要はありません. むしろ, 本書でテキストの統計モデルの基礎について一通り理解して道を作った上で, あらためて機械学習や統計学の本を読むことで, それらがより読みやすくなり, 理解が深まることを期待しています.

人文科学や社会科学においても, 計量的な方法の必要性は増加の一途をたどっており, その多くの場合にテキストの分析が必要になっています. 今までは, 統計が必要になるのは主に経済学を中心とした社会科学で, そこでは価格のような連続量が中心となっており, 従来の統計学をそのまま適用することができました. これに対して, テキストのような**離散データ**が人文科学および社会科学の両方で必要になってきたのは, Web の発達によって電子テキストが容易に入手できるようになった比較的最近のことといえます. 筆者ももとは文科系ですので, そうした分野の重要性はよく理解しているつもりです. 本書をきっかけに, 自分の手でテキストの統計的なモデル化と分析ができるようになっていただければと考えています.

また, 本書はテキストを対象にしていますが, こうした離散データに対する方法論は言語だけでなく, **他の種類の離散データについても同様に適用する**ことができます. たとえば, 文書にさまざまな種類の単語が含まれている状況は, コンビニやオンラインストアで客がさまざまな商品を購入する状況 [*1] と同じで, 実際にこうした購買データを分析する協調フィルタリングとよばれる分野は, 本書で文書をモデル化するために導入するものと, まったく同じモデルを用いています. また, 音楽の楽譜は離散的な記号で, 言語のように構造を持っていますし, バイオインフォマティクスで扱う細胞内の DNA やゲノムは ATGC あるいは 20 種類のアミノ酸を文字とした, 一種の「言語」です. 他にも, 本書で紹介する離散データのための確率モデルは, 多くの分野で適用できるのではないかと考えています. [*2]

∗1　これは, データマイニングやマーケティングの分野ではバスケット分析とよばれています.

∗2　筆者はこれまでに, 言語学, 脳科学, 音声認識, 音楽情報処理, ロボティクス, バイオインフォマティクス, 政治学といった分野と共同研究を行っています.

本書の対象読者

上に述べたように, 本書はテキストの統計的なモデルに関心のある, 広く人文科学・社会科学および理系の方, および言語の数学的なモデル化に興味を持つ一般の方を対象にしています. 確率の基礎から始め, n グラムモデルとは何か, Word2Vec は数学的には何をしているのか, テキストをどうクラスタリングすればよいのかなど, 原理的な内容を一から解説します.

本書の特徴は, 数式による定義を天下りに与えるのではなく, できる限りその導出や意味について解説していること, そしてブラックボックスのパッケージに頼らない, ということです. かわりに, テキストをモデル化する自然言語処理の数理について丁寧に説明し, 読者が自分の手で統計的な分析を使いこなせることを目標としています. たとえばパープレキシティ一つをとっても, 定義の数式をただ与えるのではなく, 基礎となる自己情報量の説明から始め, どうしてパープレキシティを考えるのか, どういう意味を持っているのかを丁寧に解説するようにしました. テキストを扱う既存のパッケージでは, 本書で紹介するような数学的な分析はほとんど行うことができませんが, それでもよいという方は多数の本がありますので, そちらをご参照ください. ただしもちろん, 普段はパッケージを使った分析を行っている方にとっても, 本書でテキスト分析の背景や理論について理解しておくことは, 適切な分析を行う上で大きな助けになるでしょう.

必要となる数学について

本書はテキストの統計的なモデル化に関する本ですので, 数式を用いることはどうしても必要になります. むしろ, 言語のようなアナログな対象をどう数学的にモデル化していくかが, 本書の主題といってもいいでしょう. ただし, 必要になるのは数 III までの高校数学と, 大学教養の数学 (解析と線形代数) の一部だけです. 本書に限らず, 統計学の理解には, 数 III までの知識はどうしても必要になります (e^x の微分・積分や対数の微分など). 現在はこのような場合のために, 多くの優れた参考書が出版されていますので, 必要な方は自習しておくようにしてください. といっても, 難しい問題が解ける必要はなく, 教科書レベル

の内容が理解できていれば充分です. 筆者自身の数学も, 数 III 以降はこうして自習したものです.

「物事は, それを生み出したのと同じ思考のレベルでは解決できない」という言葉がありますが, 筆者は言語の問題は, 言語だけを使うことでは解き明かせないと考えています.

深層学習との関係について

本書では, 深層学習に関してはもっとも基本的な単語埋め込み, 文埋め込み, 文書埋め込みまでの手法と, その背後にある数理について解説しました. 現在の自然言語処理は, こうした埋め込みを基礎とした Transformer (BERT) のような深層学習の手法なしには語ることができず, 深層学習は現在, 巨大な学習データをもとに長足の進歩を遂げています. しかし, それらは実装して動かせても, なぜ言語をうまく扱えるのかという理論的背景についてはほとんどわかっておらず, ほぼブラックボックスの状態です [*3]. そこで本書では, Transformer や LSTM といった手法についてはすでに多数の本があることを踏まえ, あえて紹介しないことにしました. ただし, 本書で説明している単語埋め込み, 文埋め込み, 文書埋め込みの範囲, あるいはそれ以外の確率的手法を用いても, 非常に多くの自然言語処理が可能なことに注意してください. 現代の深層学習は, すべてこうした教師なし学習の上に築かれたものです. Transformer のような手法は非常に多くのデータと莫大な計算時間を必要としますが, すべての自然言語処理の問題がそれで解決するわけではなく, より深い応用には, 本書で説明したような基礎知識と組み合わせることが必要になります. 本書の知識を基礎にした上で, 深層学習手法の使い方については多くの書籍が出版されていますので, 3 章の文献案内も参照してください.

---

*3　これらが内部で何を行っているのかについての研究も一部では進みつつあり, 筆者も, LSTM の内部状態が文法構造の再帰の深さをほぼ離散的に数えている[1]といった研究を行っています. 執筆時での Transformer の内部動作に関する知見は[2]にまとまっていますが, 断片的な事実の集積といえ, Transformer がなぜ言語を理解できるのか, どのように理解しているのかを説明する統一的な理論はまだ知られていません.

実装とサポートサイトについて

本書では, Python 言語を使って, 実際のテキストでの計算例を示しました. 統計の分野では R 言語がよく使われていますが, R は実行速度が遅く, 標準的にサポートされているデータ構造も少ないため, 大量のデータを高速に扱う必要のあるテキストの処理にはあまり向いていないからです. [*4] Python 言語自体の説明はしていませんので, 標準的な使い方は理解していることを前提としています [*5]. ただし, 実装部分をすべて理解できなくても, 本文の記述は追えるように配慮したつもりです. 本書で用いた例や実装はすべて, 本書のサポートサイト

`http://www.ism.ac.jp/~daichi/textmodel/`

で公開しています. また, Github のレポジトリ

`https://github.com/daiti-m/textmodel/`

でも同じファイルを公開していますので, コマンドラインから

```
% git clone https://github.com/daiti-m/textmodel
```

を実行すれば, 本書で用いたスクリプトやデータをすべてダウンロードすることができます.

実験で用いているテキストデータはすべて自由に入手できるもので, 上記のサポートサイトおよびレポジトリの “data” フォルダ

`http://www.ism.ac.jp/~daichi/textmodel/data/`
`https://github.com/daiti-m/textmodel/data/`

から入手することができます. サポートサイトには更新情報やリンク集など, 有用な情報を載せていく予定ですので, 合わせてご覧いただければ幸いです.

---

*4 不可能ではなく, 実際に政治学方法論におけるテキスト分析のパッケージである quanteda は R 言語上のパッケージです. ただし, 大量のテキストに対して重い計算を高速に実行するには, Rcpp のような外部言語の助けが必要になります.

*5 参考書を使わなくとも, オンライン上の情報で Python の基本は充分にマスターすることができます. 個人的には, M.Hiroi さんの「新・お気楽 Python プログラミング入門」`http://www.nct9.ne.jp/m_hiroi/light/index.html#python_abc` は大変お薦めで, これを読めば, 本書に必要な基本は容易に理解できるでしょう. `https://utokyo-ipp.github.io/` では東京大学の, `https://repository.kulib.kyoto-u.ac.jp/dspace/handle/2433/285599` では京都大学の Python プログラミング入門の実習テキストが, それぞれフリーで公開されています.

本書の記法

x	英小文字：変数
N	英大文字：定数またはデータ
$\mathbf{x}$	英小文字の太字：ベクトル
$\mathbf{X}$	英大文字の太字：行列
$\mathcal{L}$	カリグラフィー体：集合 (データセットや語彙など)
$\mathtt{a}$	タイプライター体：文字 (アルファベットの場合)
$\overrightarrow{w}$	単語 w の単語ベクトル
$\mathbb{E}$	期待値
$\mathbb{V}$	分散
$\mathbb{R}$	実数
$\mathbb{I}$	指示関数 ($\mathbb{I}()$ の中が真なら 1, 偽なら 0 を返す関数)
$\mathbf{I}$	単位行列
$\mathbf{0}$	ゼロベクトル (要素がすべて 0 のベクトル)
$\overrightarrow{x} \cdot \overrightarrow{y}$	ベクトル $\overrightarrow{x}$ と $\overrightarrow{y}$ の内積
^{T}	ベクトルおよび行列の転置　($\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T\mathbf{y}$)
$\langle \cdots \rangle_p$	確率分布 p による期待値
$\exp(x)$	e^x の別記法
$\propto$	比例する
$\square$	証明終わり

確率分布の略記

$\mathcal{N}$	ガウス分布 (正規分布)
Po	ポアソン分布
Be	ベータ分布
Ga	ガンマ分布
Mult	多項分布
Bernoulli	ベルヌーイ分布

目次

* の節はやや高度な内容のため, 初読の際にはスキップしてもかまいません.

1　テキストと言語のモデル化

1.1　言語とテキストの特徴

われわれは日々, テキストに囲まれて暮らしています. 図 1.1 のように, 街中にはテキストがあふれていますし, 毎日見る携帯電話の画面から Web ページ, 電子メール, 小説や雑誌に至るまで, われわれがテキストを目にしない日はないと言ってよいでしょう.

テキストは音声を書き起こしたものですが, もちろん, 言語は書き言葉から生まれたわけではありません. よく知られているように, アイヌ語は文字を持ちませんでしたし, 南米のインカ帝国でも同様でした. しかし, 最初に古代メソポタミアで文字が発明されて言葉が書き表されるようになったことで, 言語は音声によるその場限りのコミュニケーションの媒体から, 空間および時間を超えて伝わる情報を表せるようになり, 抽象化された書き言葉も出現して, 今日みられる高度な文明の礎となりました. 言語にはこのように, テキストに書き表される以前に音声や抑揚, 画像としての文字といった側面もありますが [*1], 本書では記号化されて文字として表されたもの, すなわちテキストを対象とすることにし

図 1.1: われわれは日々, テキストに取り囲まれて暮らしています.
(香港にて, 筆者撮影)

*1　こうしたテキスト以前の音声や文字画像の情報をどうテキストと組み合わせるかは, たいへん興味深い問題です. 実際に深層学習を用いて, フォントから取り出した漢字の部首の画像を文字の情報と組み合わせる研究は現在, さまざまに行われています.

ます.

テキストが, 客観的にみてほかの一般的なデータと異なっている点は何でしょうか. それは

離散的であること

かつ,

1 次元の時系列として表されていること

だと考えられます.

最初に述べたように, テキストはもともと音声を書き起こしたもので, 音声信号は連続的な空気の圧力の違いが時系列で耳に伝わってくるものです. しかし, この連続値の時系列を, われわれは “good morning” のような離散的なテキストとして認識します. 離散的とは, 情報は文字が単位の場合は “a”,…,“z”, 単語が単位の場合は “good”, “morning”, “evening”, … のように, 集合のどれかの要素として認識される, ということです. この際, 連続値である音声の周波数や画像の色とは異なり, 綴りが似ているからといって意味が似ているとは限らず, “cat” (猫) と “cut” (切る) はまったく違う言葉になっています. こうした言葉の種類は言語の場合は非常に多く, 文字でも漢字圏なら数千以上, 単語の場合は数万種類を超えるため, このあとで行うようにそれぞれの言葉をベクトルの次元とみなせば, **超高次元**になるのが普通です.

また, 画像では色の配置は 2 次元的ですが, テキストはもともと耳で聴く音声を書き起こしたものであるため, 本質的に 1 次元なのが特徴です. よって, ある言葉が次の行にある言葉とたまたま近くにあっても, 両者は原則的に無関係で, これは画像のような空間的なデータとの大きな違いです. いっぽうで言語は 1 次元ではあるものの, 句構造や埋め込み文, リズムや談話構造といった**見えない構造**が実は豊富に存在していることも特徴といっていいでしょう.

こうした離散的な時系列であるというテキストの特徴により, 連続値であることを暗に仮定した [*2] 主成分分析, 多変量解析などの古典的な統計学や機械学習の手法は, 本来はそのままでは使うことができません. それではどうすればよいのかについて, 本書で一緒に考えていきましょう.

---

*2 多くの古典的な統計理論は, 誤差が連続値でガウス分布に従うといったことを仮定しています.

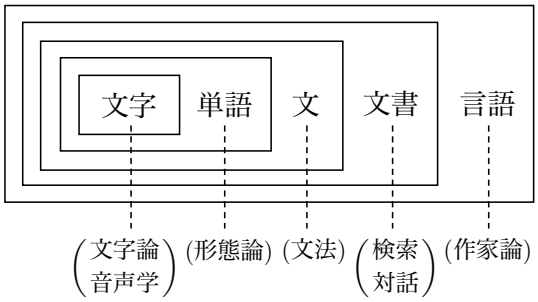


図 1.2: 言語単位の階層構造. ( ) の中に, それぞれを扱う分野の例を示しました.

1.2　テキストの階層構造

先ほど, テキストには文字と単語があると書きました. よく考えると, テキストにはもっと多くの構造があることがわかります. この全体像を図 1.2 に示しました.

(1) 第 1 のレベルは, 文字です. これは視覚から知覚する際は, 画素 (ピクセル) の集合とみることができます. 音声として聴覚から視覚する場合は, 音素になります.

(2) 第 2 のレベルは, 単語です. これはテキストでは文字の集合で, 音声の場合は音素の集合です [*3].

(3) 第 3 のレベルは, 文です. これは単語の集合で, 単語が連なって一つの文を作ります.

(4) 第 4 のレベルは, 文書です. これは文の集合で, 文が連続して, まとまった意味を持つ一つの文書となります. [*4]

(5) 第 5 のレベルは, (狭い意味での) 言語です. 多くの文書が日本語や英語など特定の言語で書かれていることを考えると, それらの文書の集合が, 日本語や英語のテキスト全体になります.

---

*3　フランス人の子供が “beaucoup (ボク)” のような難しい綴りを後から覚えるように, 文字を介さずに単語を認識することも原理的に可能です.

*4　この「文書」のことをテキストとよぶことも多いのですが, 本書では書かれた言葉全体をテキストと呼ぶことにします.

もちろん, これらの中間のレベルもあり, たとえば接頭辞や接尾辞, 語幹といった形態素を扱う形態論は文字と単語のレベルの中間にあり, 名詞句や動詞句, 係り受けといった構造は単語と文の中間に, 段落や談話構造は文と文書の中間に存在するでしょう. ただし多くの場合, 単語, 文, 文書の構造はほぼ [*5] 自明に与えられているため, 上記の分類を採用しています. このそれぞれを扱う分野の例を, 図 1.2 の下段に示しました.

重要なのは, これらは**階層構造**をなしているということです. すなわち, 文字の集合が単語に, 単語の集合が文に, 文の集合が文書に, 文書の集合が言語になっています [*6]. このうち, どのレベルに注目するかは興味と目的によるでしょう. 形態論に興味のある場合は文字がベースに, 同意や補足といった談話構造に興味がある場合は文がベースになると考えられます. [*7] エンジニアの方が文書を扱う場合も, 一般的には単語または文をベースにして文書を考える必要があるでしょう.

そこで本書では, 文字→単語→文→文書の順にその統計モデルを見ていくことにします. 必要になる統計的な概念についても, 基本的なレベルから少しずつ学んでいくことにしましょう.

1.3 教師あり学習と教師なし学習

言葉を計算機で扱う自然言語処理 [*8] は, **機械学習**の一種と考えられますが, 一般に機械学習は, 教師あり学習と教師なし学習 (および強化学習) に分けることができます.

∗5 正確に言えば, 「単語」や「文」とは何かというのは自明ではない問題です. 日本語では空白で区切られた「単語」は存在しませんし, 「。」で文の終わりが示されるものの, 必ず「。」で終わっているとは限らず, 口語の場合はどこまでが文かも曖昧なところがあります. この問題については, 4.1 節を参照してください.

∗6 言語学では, 文は形態素の連続に, さらに語は音素の連続に分かれるという構造は**二重分節**とよばれています[3].

∗7 もちろん, 形態論は前後の単語の影響を受けますし, 談話構造は文に含まれる単語を考慮する必要があるでしょう. このように, 隣接するレベルの情報は必要になりますが, 隣接しないレベルは多くの場合必要ないでしょう. すなわち, 形態論には文書の構造は関係せず, 談話構造には文字のレベルはほとんど影響しないと考えられます.

∗8 「言語」にはオートマトン理論などで使う形式言語やプログラミング言語なども含まれますので, 実際に人間が使う言語を特に自然言語とよびます. これらの間には深い関係があります[4].

```
* 0 1D
冷戦 れいせん * 名詞 普通名詞 * *
の の * 助詞 接続助詞 * *
* 1 9D
終えん しゅうえん * 名詞 サ変名詞 * *
が が * 助詞 格助詞 * *
* 2 3D
「 「 * 特殊 括弧始 * *
どんな どんな * 指示詞 連体詞形態指示詞 * *
* 3 5D
政党 せいとう * 名詞 普通名詞 * *
でも でも * 助詞 副助詞 * *
* 4 5D
政権 せいけん * 名詞 普通名詞 * *
を を * 助詞 格助詞 * *
：
```

(a) 品詞分析 (形態素解析) のための教師データである京大コーパス[5]の一部. 京大コーパスには人手で準備した 38,000 文のこうした「正解」があり, 通常の形態素解析器はこれを教師データとして, 単語の境界や品詞を学習しています.

majQa'. ruv Suqpu' ghaH. tar qeqta' ghaH. 'a DaH manoDchuqQo'jaj, *Hamlet*. vavwI' HoH, HoHwIj je pIch Hoch vIqIl. 'ej HoHlIj pIch yIqIlneS je.

(b) このように人間が教師データを作れない「宇宙人語」[*10] の場合は, 自然言語処理は何もできないのでしょうか?

`livedoor-homme` 次世代への願いを込めた燃料電池電気自動車ガソリンと電気モーターによるハイブリッドシステム車は、自動車メーカー各社が開発・販売に力を入れたことにより、ここ数年で随分と認知され、シェアも獲得している。さらには..

(c) 5 章で使う livedoor コーパスの一部. 明らかに車の話をしていますが, 教師ラベルは `livedoor-homme` としかつけられていません. 車を話題にしているテキストだけを抽出するには, どうしたらよいのでしょうか.

図 1.3: 自然言語処理での教師あり学習とその限界.

教師あり学習の目標は, 入力 $\mathbf{x}$ に対する出力 $\mathbf{y}$ を正しく求めることです. 2 章で学習する条件つき確率を使うと, $p(\mathbf{y}|\mathbf{x})$ を求めることに相当します. 自然言語処理の場合の入力 $\mathbf{x}$ は多くの場合は文や文書で, 出力 $\mathbf{y}$ はその文の翻訳や対話の応答, 文書のカテゴリなどになります. この学習には, 人手で準備した大量のペア $(\mathbf{x}, \mathbf{y})$ を必要とします. これに対して**教師なし学習**では, 言語, すなわち入力 $\mathbf{x}$ 自体をモデル化します. 確率の言葉では, $p(\mathbf{x})$ を求めることになります. これには何かモデルが必要ですから, 実際には見たい構造 $\mathbf{z}$ を未知の**潜在変数**として設定し, モデル $p(\mathbf{x}, \mathbf{z})$ を考えることになります [*9].

この二つの違いを理解するために, 本書の 4 章でも扱う品詞分析を考えてみましょう. 教師あり学習では, $\mathbf{x}$ = “She is a girl” のような入力に対する “正解” の品詞列 $\mathbf{y}$ = “名詞–動詞–冠詞–名詞” のような $(\mathbf{x}, \mathbf{y})$ のペアを, 学習データとして人手で大量に (たとえば数万個) 準備します. この例を, 図 1.3(a) に示しました. 学習の目標は, 学習データにない新しい入力 $\mathbf{x}$ = “Andy sings well” のよ

*9 可能な $\mathbf{z}$ について周辺化すると, これは $p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}, \mathbf{z})$ を求めていることになります.

*10 これは実は, 4 章で分析する『スタートレック』のクリンゴン語で, 原文は『ハムレット』第五幕の “*He is justly served; It is a poison tempered by himself. Exchange forgiveness with me, noble Hamlet. Mine and my father's death come not upon thee, Nor thine on me!*” です.

状態 1		状態 2		状態 3		状態 4		状態 5		状態 6	
she	432	the	1026	was	277	and	466	way	45	little	92
to	387	a	473	had	126	of	343	mouse	41	great	23
i	324	her	116	said	113	in	262	thing	39	very	22
it	265	very	84	be	77	said	174	queen	37	long	22
you	218	its	50	is	73	to	163	head	36	large	22
alice	166	my	46	went	58	as	163	cat	35	right	20
and	147	no	44	were	56	that	125	hatter	34	same	17
they	76	his	44	see	52	for	123	duchess	34	good	17
there	61	this	39	could	52	at	122	well	31	white	11
he	55	an	37	know	50	but	121	time	31	other	11
that	39	your	36	thought	44	with	114	tone	28	poor	10
who	37	as	31	herself	42	on	83	rabbit	28	first	10

図 1.4: 『不思議の国のアリス』で学習した HMM (4 章) の潜在状態と, 割り当てられた単語の回数. 「主語」「動詞」「形容詞」といった概念が, **教師なし**で自動的に学習されています.

うな文について, 正しい品詞列を出力することです. このことの利点は, 常に人間の想定に沿った出力が得られることです. 一方で欠点は, **決して事前の想定内を出られない**ということです. 入力文が学習データにない絵文字で終わっていたとき, できるのは既存の品詞の中から無理矢理どれかを選ぶことだけでしょう. また, 見たことがない綴りの単語への対応も難しくなりますし, 文書のカテゴリも図 1.3(c) のように, 事前に設定した中から選ぶだけしかできません.

これに対して教師なし学習では, 図 1.3(b) のような入力文 $\mathbf{x}$ の裏に未知の状態系列 $\mathbf{z}$ があると考え, $p(\mathbf{x}, \mathbf{z})$ が高くなるように $\mathbf{z}$ を学習します. うまく学習すれば, ただ**単語列を与えただけ**で, 図 1.4 のように**実質的に「品詞」とみなせる潜在状態**を推定することができます. このことの利点は, 常に言語 $\mathbf{x}$ だけを見ているので, 適切な統計モデルを作れば, **言語自体から学習する**ことが可能だということです. 上記のような絵文字が文末で特別な意味を持つと統計的に判断できれば, これに新しい状態が割り当てられますし, 新しい言葉への対応も容易です. 原理的には, 未知の言語を扱うこともできます. 逆に欠点は, 一般に学習や評価が難しいこと, また学習された状態が人間の望む「品詞」と一致するとは限らないということです. たとえば, 日本語に “形容動詞” を認めるかについては言語学者の間にも諸説ありますが, モデルが特定の立場で出力してくれる

とは限らず, 解釈も別に必要になります. [*11]

この後で学習するように, 実はこの二つのモデルは掛け合わせると $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x})\, p(\mathbf{x})$ となり, $\mathbf{x}$ と $\mathbf{y}$ の同時確率を与えますから [*12], 本来はこの両方が必要です. 特に, 一部しか正解 $\mathbf{y}$ が与えられない**半教師あり学習**のためには, 教師なし学習のモデルが必要になります. 多くの教科書が教師あり学習を主に扱っていることから, 本書ではより難しい問題である, 教師なし学習に焦点を当てて解説することにします. [*13]

1.4　統計的な方法とヒューリスティックな方法

ここで, 本書がどうして「統計的」なテキストのモデルを考えるかについてふれておくことにします. たとえば, 本書の 5 章では文書のモデル化を考えますが, ある文書が 20 個の単語からなり, 含まれる単語の頻度がそれぞれ

“チューリップ” が 4 回, “栽培” が 3 回, “の” が 8 回, “こと” が 5 回

だったとしましょう [*14]. この文書を本書のように確率的なモデルを考えず, 単に各語彙を次元とみて頻度を並べたベクトル

$$\mathbf{x} = (0, \cdots, 0, 4, 0, 3, 0, \cdots, 0, 8, 0, \cdots, 0, 5, 0, \cdots, 0)$$
$$\qquad\qquad\quad \uparrow \quad \uparrow \qquad\quad \uparrow \qquad\quad \uparrow$$
$$\text{チューリップ} \quad \text{栽培} \qquad \text{の} \qquad\quad \text{こと}$$

で表せばよい, という人がいるかもしれません. [*15]

この場合, $\mathbf{x}$ は語彙数 (たとえば 10000) の次元をもつ, 超高次元のベクトルで, そのほとんどは 0 です. あれ, これでは文書の長さで表現が変わってしまうの

---

*11　この他に, BERT の学習などでも使われている**自己教師あり学習**があります. これは観測された言葉を正例, それ以外のランダムな候補を負例として教師あり学習を行うもので, 人手によるラベルは使っておらず, 二者の中間とみなすことができるでしょう.

*12　通常は教師なしデータ $\mathbf{x}$ の方が圧倒的に多いため, ナイーブにこれを行うと $\mathbf{y}$ が軽視されてしまいますので, 実際にはもう少し検討が必要です[6, 7].

*13　この二者の立場の違いは, 事実上, 工学と理学の違いといってもいいでしょう. 教師なし学習は, 自己組織化[8]の考え方の統計的な表現ともいえます.

*14　これでは日本語になりませんが, 説明のために簡単にしています.

*15　実際にこれは「ベクトル空間モデル」とよばれ, 初期の自然言語処理や情報検索で使われていました[9].

で, ベクトルの長さを 1 にする [*16] ために $\sqrt{4^2+3^2+8^2+5^2}=\sqrt{114}$ で割って,

$$\mathbf{x}=\left(0,\cdots,0,\frac{4}{\sqrt{114}},0,\frac{3}{\sqrt{114}},0,\cdots,0,\frac{8}{\sqrt{114}},0,\cdots,0,\frac{5}{\sqrt{114}},0,\cdots,0\right)$$

とした方がいいですね. あるいは, “の” や “こと” のような一般的な単語の重みを下げるために, 各単語の頻度を tf.idf (5 章) で置き換えた方がいいでしょうか. こうしてベクトルが得られたら, あとは K 平均法でクラスタリングしたり, 主成分分析にかけて次元圧縮し, 「主成分」を抽出すれば, 一丁あがりです. …

こうした素朴な方法は, 特に文系や初心者を対象とした「テキストマイニング」の本でよくみられますが, 何が問題なのでしょうか.

問題は色々ありますが, まず第一に, 上のような処理をすることに**直感以外の理由がなく, 処理の判断基準がない**, ということが挙げられます. 上でベクトルを正規化するのに, 和を 1 にする方法と長さを 1 にする方法がありましたが, どちらを使えばよいのでしょうか. また tf.idf には, tf 部分を対数にするのか (するなら, 対数の底を何にするのか), tf に 1 を足すのか, といった多くのバリエーションがあります. そもそも tf.idf が最適だという保証もありませんし, これらの選択によって, 分析の結果はまったく違ってきてしまいます. 一方, 統計モデルを使えば, **テキストの背後にある数学的なモデル**が明確になります. その際, 本書の 5 章でみるように数学的に自然な形で “の” のような一般的な語の重みを自動的に下げることができ, ベクトルを正規化する必要もありません. 実際に自然言語処理の歴史は, 最初は発見的に導入された手法が, より性能の優れた数学的な手法に置き換えられることの連続でした.

第二に, よく行われる主成分分析やベクトルのクラスタリングは, **本来連続的なデータを対象にしており, 頻度のような離散的なデータに直接適用してはいけない**, という点が挙げられます. 上の文書ベクトルは超高次元 (たとえば 10000 次元) な上, 値は必ず正で, そのほとんどは 0 になっている, という制約があります. 通常の主成分分析は値が負の場合もある連続値で, 誤差が平均 0 のガウス分布に従うことを仮定しており [10], こうした状況には当てはまりません. 単

*16 頻度の総和で割って正規化することもできますが, そうすると値は, その単語の文書内での確率と同じです. それならば, なぜ最初から確率的に考えないのでしょうか?

表 1.1: 各テキスト A,B,C,D に現れた単語とその頻度の例.

テキスト	A	B	C	D
単語 1	0	6	20	0
単語 2	2	8	30	8
単語 3	0	1	12	1
:	:	:	:	:
単語 V	3	0	5	2
長さ	50	200	1000	500

語の頻度も充分に高ければ, 近似的に連続値とみなすことができますが, これは頻度の高い, ごく一部の単語についてしか成り立ちません. 言語では数回といった低頻度でも意味があることが多く, たとえば, ある人物が「開腹」という言葉を数回使っただけで, われわれはこの人は医師あるいは医療系の方だという大きな情報を得ることができます. しかし, こうした低頻度の言葉を, 通常の多変量解析で適切に扱うことはできません.

たとえば, アンケートの自由回答やある小説家の作品群といった複数のテキストを比較するために, 表 1.1 のように単語の頻度を数えたとしましょう. この表は, 単語 1 はテキスト A には 0 回, B には 6 回, C には 20 回, …出現したことを表しています [*17]. このとき, テキストを比較してクラスタ分析を行うために, テキスト A の特徴ベクトルを, 表 1.1 を縦に読んで $(0, 2, 0, \cdots, 3)$, C の特徴ベクトルを $(20, 30, 12, \cdots, 0)$ などとするのは, 明らかに適切ではありません. なぜなら, そもそも表 1.1 の背後にある各テキストの長さが大きく違っているからです. テキスト C は長いので頻度は自然と大きくなり, このままではテキスト C の影響が過大に見積もられてしまいます [*18]. また, テキスト D は表 1.1 で見えている範囲の出現頻度はテキスト A と似ていますが, もとになるテキストはずっと長いため, 頻度の意味はかなり違っているはずです.

明らかに, この場合は頻度そのものではなく, 各テキストの中でそれぞれの単語が出現する**確率**を考えるべきでしょう. 表 1.1 の頻度を各テキストの長さで

∗17　政治学方法論の分野でテキスト分析のために広く使われている R のパッケージ `quanteda` では, こうした表を簡単に作ることができ, 基本的なデータ構造になっています. また, 仏教学の分野で石井公成氏が開発した NGSM [11]は, 仏典の漢字 n グラムについてこうした表を作成して分析を行うものです.

∗18　一般にアンケートの自由回答などでは, テキストの長さは大きく違っているのが普通です.

表 1.2: 表 1.1 を, 各テキスト内で単語が出現する**確率**に直したもの.

テキスト	A	B	C	D
単語 1	0	0.03	0.02	0
単語 2	0.04	0.04	0.03	0.016
単語 3	0	0.005	0.012	0.002
:	:	:	:	:
単語 V	0.06	0	0.005	0.002

表 1.3: さらに確率の負の対数をとることで, **情報量**に変換したもの.

テキスト	A	B	C	D
単語 1	∞	3.51	3.91	∞
単語 2	3.22	3.22	3.51	4.14
単語 3	∞	5.30	4.42	6.21
:	:	:	:	:
単語 V	2.81	∞	5.30	6.21

割って求めた確率は, 表 1.2 のようになります. これで, テキストの長さにかかわらず, 単語の出やすさを平等に比較することができました.

ただし, これで終わりではありません. 単語 1 がテキスト B と C で出現する確率はそれぞれ 0.03 と 0.02 で, 0.01 の差があります. 1.5 倍の差ですね. 一方で単語 3 が C と D で出現する確率は 0.012 と 0.002 で, これも 0.01 の差ですが, 確率が小さいので実は 6 倍の差があるわけです. これを同じ違いとするのは, 不合理ではないでしょうか [*19]. こうした場合は表 1.3 のように, 確率の負の対数をとって本書の 2 章で説明する**情報量**に直すことで, 違いを適切に表すことができます. 表 1.3 のように, このとき前者の差は $-\log 0.03-(-\log 0.02)=3.51-3.91=-0.41$, 後者の差は $-\log 0.012-(-\log 0.002)=4.42-6.21=-1.79$ となり, 同じ確率 0.01 の差でも, 後者の方が大きな意味をもつことを表すことができました. 実際にはさらに, ∞ を避けたり, 単語 2 のようにすべてのテキストで高確率で出現する単語 (「の」など) の重みを下げるために, 3 章で説明するように確率の平滑化を行ったり, 非負の自己相互情報量 (PMI) を計算して特徴量とするのがよいでしょう. いずれにしても, こうした考察には, 本書で説明するような**確率・統計的な知識**が不可欠になります.

第三に, 最初に示したような**ヒューリスティックな方法は理論的な裏付けがないため, 方法を拡張することができない**という問題もあります. たとえば, 単語の頻度に外れ値がある場合や, 観測されない欠損値がある場合はどうすればよいのでしょうか. 文書をベクトル化したとして, その時間的な変化を見たい場合も, 理論的な裏付けがなければ, 「適当にプロットする」程度のことしかできま

*19 頻度は確率に比例しますので, 頻度を直接使うもとの方法も, 同じ問題を持っていることがわかります.

せん．これには無数の任意性がある上に，たとえば観測のなかった時期があると，一気に使えなくなってしまいます．こうした例は，統計モデルを導入すれば，すべて自然な形で解くことができます[*20]．

1.5　本書の概要と読み方

本書で確率・統計的な手法を学ぶと，何ができるようになるのでしょうか．ここでは本書の構成を紹介しつつ，この後に出てくる解析例をいくつか紹介します．これらの結果のほとんどは，**深層学習のブラックボックスをむやみに回しても不可能なもの**であることに注意してください．なお，基礎についてはすでに知っている読者の場合でも，本書では実際の例や脚注などを通じて，面白く読めるように配慮して執筆しました．本書を読むための全体のフローチャートを，図 1.5 に示しました．

先に説明したように，本書では文字・単語・文・文書の順でテキストの統計モデルについて説明します．確率の基礎から始まって，自然言語処理の最先端の内容までが含まれていますので，**一度で全部を理解できる必要はありません．**最も基本的な話が先になるように配置していますので，2 章からわかる範囲で読み進めてゆけば，基本を理解することができます．特に，文 (4 章) および文書 (5 章) の章はある程度独立に読めますので，興味のある部分から読み始め，必要に応じて前に戻ってもよいでしょう．

2 章では，文字の統計モデルを入り口に，確率と統計モデルの基礎について説明します．同時確率や条件つき確率，ベイズの定理などについて学び，統計モデルの評価についても説明します．文字 n グラムモデルを作ると，たとえば図 1.6

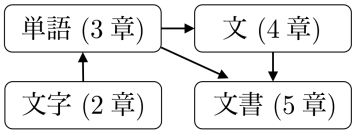


図 1.5: 本書の読み方のフローチャート．

*20　この場合，統計モデルを使えば，5 章で説明するような別の数学的なベクトル化を行った上で，カルマンフィルタやガウス過程[12]で時間発展を追いかけることも可能になります．

```
mass, moushly, toung, loop, cright, ked, whenneadfult,
scriontict, leanquit, but, tring, hemple, foothed, wroar
```

図 1.6: 『不思議の国のアリス』の語彙をデータにして, 文字 n グラムモデルからランダムに生成した単語の例. ($n=3$)

のように英語の新しい単語をランダムに生成することも可能になります. 情報量やエントロピーの意味についても, ここで学びましょう.

3 章ではそれを基礎に, 単語の統計モデルについて説明します. 文字よりも圧倒的に (原理的には無限に) 種類がある, 単語の場合について有用な冪乗則や確率の平滑化について学んだ後, ニューラル単語ベクトルの学習とその性質について学びます. Word2Vec は数学的には何を学習しているのか, 単語ベクトルの「引き算」はなぜ成り立つのかといった原理について説明します. 日本語の場合, 学習された単語ベクトルを 2 次元に可視化すると, たとえば図 1.7 のようになります. これらの間にはどんな関係が成り立っており, それはなぜなのでしょうか. 言語のベイズ的なモデル化に不可欠なディリクレ分布やポリア分布についても, この章で学習します.

4 章では単語が連なった, 文の統計モデルについて説明します. 本章で解説する数学的な文ベクトルを使うと, コーパスから意味的に類似した文を簡単に計算することができます (これには複雑な深層学習は不要です). また, 文字や単語などの系列を扱う隠れマルコフモデル (HMM) とそのベイズ学習について説明し, この例を通じてマルコフ連鎖モンテカルロ (MCMC) 法の一種である Gibbs サンプリングについて学びます. HMM を使うと, たとえば単語の背後に隠れた「品詞」を人間の主観を交じえず, 統計的に教師なし学習することが可能です. 図 1.4 は, 『不思議の国のアリス』の本文だけから自動的に学習された「品詞」の例でした.

5 章では, 最も実用性の高いと考えられる文書の統計モデルについて説明します. 簡単なナイーブベイズ法による分類から始め, 文書の意味的なクラスタを教師なしで発見する UM および DM, その学習のための EM アルゴリズムについて学びます. さらにその拡張として, トピックモデルとして知られる LDA (潜在ディリクレ配分法) について説明します. LDA では, 文書集合から図 1.8 のような潜在的な話題, すなわち「トピック」を完全に自動的に学習し, 各文書を

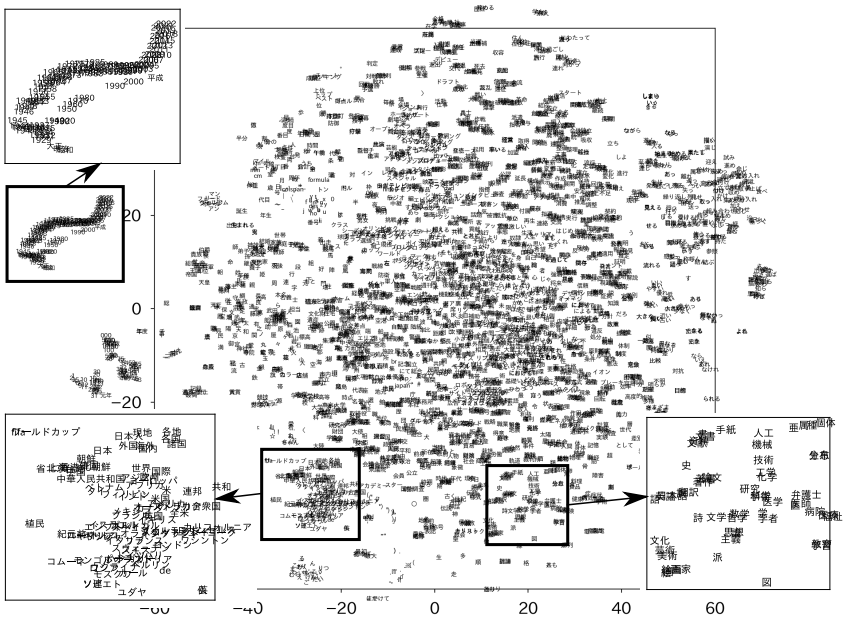


図 1.7: 日本語 Wikipedia のテキスト `ja.text9` から学習した 400 次元の単語ベクトルを, t-SNE [13]で 2 次元に可視化したもの. 意味の近い語は近いベクトルになっています.

このトピックの上の確率分布 $\boldsymbol{\theta}$ として表します. 最後に, 単語ベクトルを用いた数学的な「文書ベクトル」の計算法について学んだ後, 若干高度ですが, 筆者の研究である確率的潜在意味スケーリング (PLSS) について紹介します. PLSS では, 心理統計学の理論に基づき, テキストにある軸で測った潜在的な極性 θ を連続値として図 1.9 のように推定することができます.

本書では, 必要になる確率論や情報理論の基礎, EM アルゴリズムや MCMC 法などの推定法についても, 従来の教科書のように無味乾燥な「予備知識」の章を立てるのではなく, 必要に応じて本文の中で, 言語の例を使いながら説明することにしました. 特定の概念 (たとえばエントロピー) が何だったかな, と復習したい方は, 巻末の索引を活用してください.

なお, * **をつけた節はやや高度なため, 初読の際は飛ばしてもかまいません.**

トピック 7	
世	0.586
伯	0.566
在位	0.556
ボヘミア	0.506
辺境	0.505
皇帝	0.497
フリードリヒ	0.494
公	0.494
von	0.492
侯	0.492
妃	0.486

トピック 17	
密	0.710
真言	0.670
如来	0.630
印	0.573
口	0.560
意	0.543
身	0.543
理解	0.539
浄土	0.534
動き	0.533
救済	0.515

トピック 104	
気動	0.639
貨車	0.587
国鉄	0.559
車	0.554
JR	0.542
貨物	0.542
車両	0.533
形	0.529
幹線	0.515
運転	0.515
鉄道	0.498

トピック 248	
JRA	0.632
競馬	0.544
サラブレッド	0.538
馬	0.535
ドラフト	0.523
競走	0.516
エー	0.513
指名	0.511
赤	0.505
グリーン	0.494
解説	0.492

トピック 325	
技能	0.673
職業	0.608
能力	0.593
検定	0.587
コンクリート	0.584
合格	0.581
資格	0.571
施工	0.563
試験	0.552
ブロック	0.551
ガラス	0.551

トピック 369	
氏	0.412
藩	0.369
藩主	0.360
寺	0.359
江戸	0.357
戦国	0.350
守	0.339
時代	0.337
天皇	0.335
当主	0.332
武将	0.331

トピック 390	
大学	0.351
教授	0.333
名誉	0.287
博士	0.283
研究	0.282
卒業	0.280
工学	0.276
理事	0.273
学会	0.263
文学	0.262
会長	0.262

トピック 493	
ライブ	0.616
LIVE	0.582
Tour	0.521
DVD	0.512
映像	0.505
TOUR	0.496
Live	0.493
Blu	0.491
ray	0.489
ツアー	0.483
ドーム	0.481

図 1.8: 日本語 Wikipedia の記事の UM (5 章) によるクラスタリングをベイズ学習することで推定された, 潜在トピックの例 ($K=500$). 数字はトピックとの相関を表す NPMI (5.2.1 節) の値です. 非常に専門的なトピックが, 完全に自動的に学習されています.

* の節で説明した方法が後で参照されることもありますので, その場合は必要に応じて戻るようにしてください.

また, 本書はわれわれの周りに自然にみられるテキストをどうモデル化するかを考えていますので, 文書分類や通常の形態素解析のように, テキストにその所属するクラスや単語境界といった「正解」があらかじめ人手で付与されている場合の分類問題, すなわち教師あり学習は対象としていません.[*21] これらは多くの場合, SVM や CRF といった汎用の機械学習手法を適用すれば充分だからです. テキストに対する教師あり学習については, 2 章の文献案内を参考にしてください.

*21 確率の言葉でいえば, 通常の教師あり機械学習はテキスト $\mathbf{x}$ に付与されたラベル $\mathbf{y}$ をいかに予測するか, すなわちラベルの確率 $p(\mathbf{y}|\mathbf{x})$ だけをモデル化しており, 本書のようにテキスト自体の構造やそのラベルとの関係, すなわち $p(\mathbf{x})$ や $p(\mathbf{x},\mathbf{y})$ はモデル化していない, ということです.

極性 θ	発言者	発言内容
1.9838	玉木委員	もう一年たっているんですよ、対中国については。二〇二五年目標という..
1.9511	玉木委員	遺伝子組み換え食品は有機の対象にならないんだけれども、今大臣がおっ..
1.2069	野上国務大臣	農協改革につきましては、農業者の所得向上を図るとの原点を踏まえて、..
1.0668	池田大臣政務官	二〇三〇年の五兆円の輸出目標を達成するためには、主として輸出向けの..
0.9229	重徳委員	農林水産行政の責任を今後も追及していきたいと思います。　大臣にはし..
0.8061	玉木委員	国民民主党の玉木雄一郎です。　まず、次期作支援交付金の追加公募につ..
⋮		⋮
−0.4057	藤田委員	ありがとうございます。簡潔に言うと、経済事業とかがある特殊性みたい..
−0.7313	高鳥委員長	これにて趣旨の説明は終わりました。　次回は、来る十九日水曜日午前八..
−1.0786	高鳥委員長	これより採決に入ります。　内閣提出、農業法人に対する投資の円滑化に..
−1.0952	光吉政府参考人	お答えいたします。　収入保険は、品目の枠にとらわれず、農業経営者ご..
−1.7119	石川（香）委員	ただいま議題となりました附帯決議案につきまして、提出者を代表して、..
−3.3202	野上国務大臣	平成二十二年の開門を命ずる福岡高裁の判決が確定した後、国は開門義務..

(a) 国会・農林水産委員会の各発言に推定された発言の極性 θ (一部).

単語 v	極性 ϕ_v		
まずは	0.5167	一番	0.3982
なので	0.4560	どうにか	0.3940
一つ	0.4364	しっかり	0.3903
ミリ	0.4246	同時に	0.3897
増やす	0.4178	早く	0.3890
整い	0.4109	戦える	0.3886
もっと	0.4014	重点	0.3788
もう少し	0.4012	とにかく	0.3769
植え付ける	0.3995	歩	0.3762
切り替える	0.3985	試し	0.3681

(b) 玉木議員側 (正側) の極性の大きい単語

単語 v	極性 ϕ_v		
訴訟	−0.5769	シベリア	−0.4599
傍聴	−0.5646	退け	−0.4491
毀損	−0.5519	高裁	−0.4462
原告	−0.5481	弁護	−0.4431
敗訴	−0.5147	最高裁	−0.4410
控訴	−0.5010	紛争	−0.4303
係争	−0.4963	賠償	−0.4229
裁判所	−0.4962	在住	−0.4199
判決	−0.4808	証言	−0.4196
審	−0.4727	棄却	−0.4105

(c) 田村議員側 (負側) の極性の大きい単語

図 1.9: PLSS (5 章) によるテキストと単語の潜在的な極性の推定. 国会の議事録において, 少数の正例と負例の文書を与えれば, これらの埋め込み空間上で極性を表す軸を学習し, この軸に沿った発言の極性 θ および単語の極性 ϕ_v を, 平均 0 の連続値として推定することができます.

1.6　本書の例と実装について

本書で使っているスクリプトやデータはすべて, v ページの「はじめに」の「実装とサポートサイトについて」で示した方法で入手することができます. 無印のタイプライター体の例は, 対応する章の Jupyter Notebook にあるように, Python の Jupyter Notebook に順番に入力して実行できるようになっています. ⇒ は出力です [*22].

```
    print ("これは例です. ")
⇒ これは例です.
```

*22　Jupyter Notebook の `Out[]:` にあたります.

ただし, 複雑な例ではノートブック上での入力や実行には限界がありますので, %で始まる行はスクリプトやファイルのある各章のフォルダでコマンドラインを実行することを表しています. % から後が, 実際に実行するコマンドです.

```
% date
⇒ 2020 年 11 月 22 日 日曜日 22 時 35 分 56 秒 JST
```

コマンドラインは, MacOS や Linux では「ターミナル」アプリを起動すればそのまま使うことができます. Windows では, WSL (Windows Subsystem for Linux) を導入するといいでしょう. こうした環境の構築については, たとえば [14, 3 章]などで実行例とともに詳しく解説されていますので, そちらを参照してください.

テキストを扱う場合は, Jupyter Notebook や RStudio のような環境の中ではできることが限られるため, プログラム開発の面からも, ぜひスクリプトを書いてコマンドラインで実行できるようになっていただきたいと考えています. これにより複雑なプログラムとそのデバッグが可能になるほか, 汎用性のあるモジュールを自分で書いて `import` することで, プログラムの再利用が可能になり, より見通しのよい開発を行うことができます. テキストを扱うには豊富な標準コマンドがあり, 本書でも様々なスクリプトを書いて使っていますので, ノートブックや統合環境の箱庭の中だけで分析を行うのではなく, ぜひコマンドラインを使いこなせるようになってください.

なお, 記述を短くするため, 実行例はすべてのスクリプトやファイルがカレントフォルダにあることを前提にして書かれています. 実際には v ページのように `git clone` するとスクリプトは `bin/`に, データは `data/`などにありますので, 実行の際はすべてをカレントに移動するか, 適宜パスを補ってください. 本書は「何も考えずに実行できる」といった本ではなく, スクリプトはすべて内容を読んで使うことを前提にしています.

本書の Python スクリプトもこれが「正解」ということはなく, 様々な実装がありえます. 理解のために, ぜひ中身を読んで自分で試してみてください. その際, 各章の最後にある「演習問題」で興味を持ったものを行ってみるとよいでしょう.

1 章の文献案内

本章は導入ですので, 統計的自然言語処理と教師なし学習を扱う文献について紹介します. 統計的自然言語処理の最も有名な教科書は, 現在も最前線で活躍する研究者である Schütze と Manning による “Foundations of Statistical Natural Language Processing” (通称 FSNLP) [15] *23 でしょう. 1999 年に出版された FSNLP には, 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. 最近になり, 『統計的自然言語処理の基礎』の名前で日本語訳も出版されました[16]. 日本語の教科書としては, 北による『確率的言語モデル』[17]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. 本書はおも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, その差異も含めて最新のベイズ統計学の立場から扱うものとしました.

言語学のように言語をより細かく見る場合は, 音韻論や形態論なども議論されている “Speech and Language Processing” (通称 SLP) [18]が定評のある教科書です *24. ただし, 言語学者の Jurafsky を中心に書かれた本のため, 数学的な記述にはやや不正確なところもあります.

教師なし学習を主に扱っている教科書は大変少ないのですが, 石井と上田による『続・わかりやすいパターン認識－教師なし学習入門－』[19]は基礎からわかりやすく書かれた, たいへん優れた教科書です. 英語の資料としては, ケンブリッジ大学の Ghahramani による教師なし学習の解説[20]および講義スライド *25 を, 本書でも参考にしました.

テキスト分析のための環境設定および, 比較的初歩的な分析については『Python ではじめるテキストアナリティクス入門』[14]によくまとめられていますので, 初心者の方はこうした場所から始めて, 感覚をつかむのもよいでしょう.

∗23　https://nlp.stanford.edu/fsnlp/

∗24　2025 年現在, 第 3 版の出版が予定されており, https://stanford.edu/~jurafsky/slp3/ で全文の草稿が公開されています.

∗25　https://mlg.eng.cam.ac.uk/zoubin/course05/

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. このテキストでは, 1 行がほぼ 1 文に対応しています.

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. ↓で行の終わりを示しています.

```
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

$$p(c) = \frac{n(c)}{N} \qquad \text{(確率の最尤推定)} \tag{2.1}$$

と考えてよいでしょう. $p()$ は, 確率を表す記法です. [*4] 式(2.1)の確率は最尤(さいゆう)推定とよばれ, 現在手元にある学習データの確率 (39 ページで説明するように, これを尤度(ゆうど)といいます) を最大にする値ですが [*5], ここでは簡単に, 頻度を割り算して求める, もっとも簡単な確率の推定値だと考えてください.

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, ...
}
```

*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.

*5　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, 本当は自明ではなく, 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. 本書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[21, 1.5.2 節]などを参照してください.

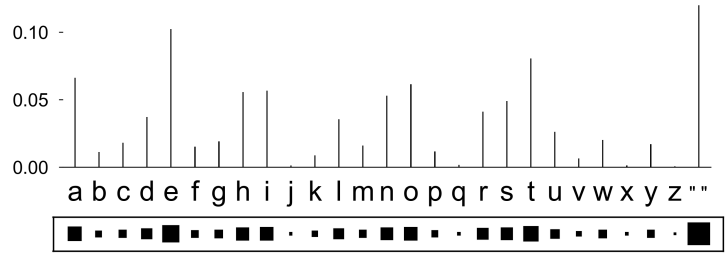


図 2.2: 『不思議の国のアリス』`alice.txt` における各文字の出現確率.

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 の各文字の下にある■はヒントン図 [*6] 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.

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, どうやって調べればよいでしょうか.

---

*6 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` を置きました.

*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.

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
  ...
  yy = 1
```

*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, 44 ページの脚注を参照してください.

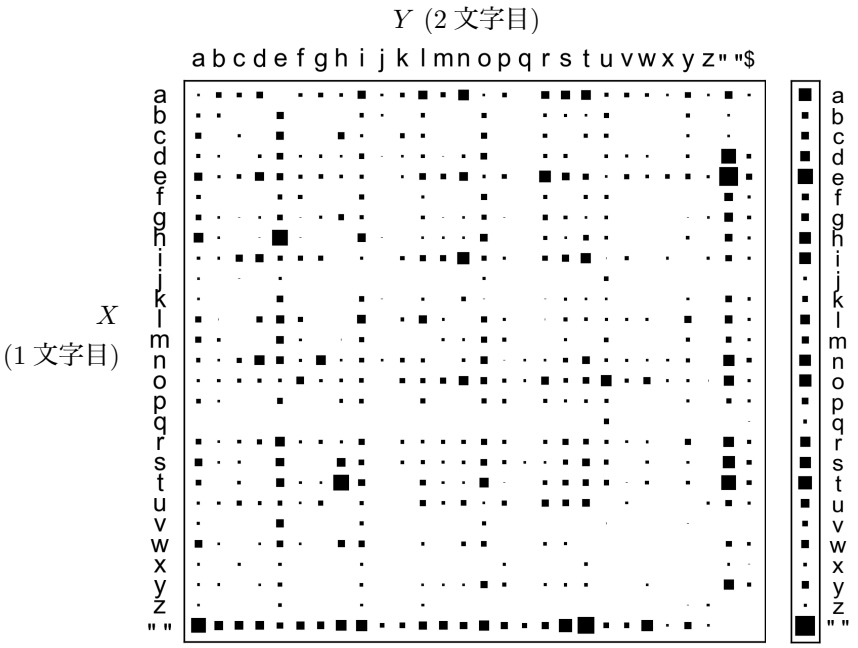


図 2.3: 『不思議の国のアリス』から計算した文字バイグラム確率 $p(X,Y)$ を表す行列 $\mathbf{P}$. 確率の総和は, 行列全体で 1 になっています. 右側に, 行列を横方向に和をとった, 各文字の周辺確率 $p(X)$ を示しました.

```
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
```

のように保存すると, 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.

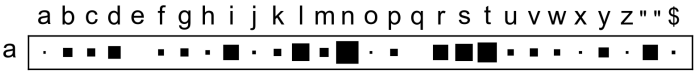
バイグラムの 1 文字目と 2 文字目を表す変数をそれぞれ X, Y とおくと, バイグラムの確率は

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

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 のように, ある確率に従って具体的な値をとる変数のことを, **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



のようになっていますが, これは

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.

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

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

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

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 の後にいろいろ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,

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

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

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

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.

実際に確かめてみましょう. 上の例では,

*9 ヒントン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.

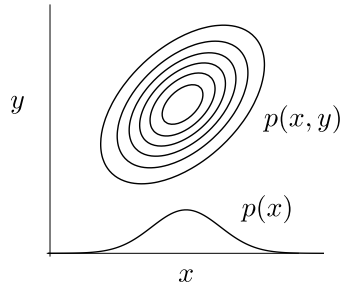


図 2.4: 同時確率密度 $p(x,y)$ の周辺化. y に関して積分して確率密度関数を下に「つぶす」ことで, x だけの関数 $\int p(x,y)\,dy = p(x)$ が得られます.

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

ですが, 式(2.2)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[12]. 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.

---

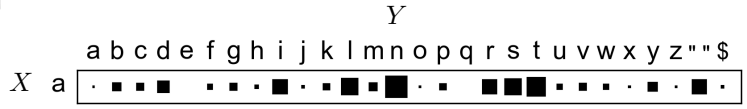
*10 ライプニッツの積分記号 $\int$ は, そもそも Sum (和) を表す S の変形であることを思い出してください. 厳密にはこれらはすべて証明が必要ですが, 本書は確率論の本ではありませんので, 立ち入らないことにします. 基礎的な定義が気になる方は, [22]や最近の[23]といった優れた教科書を参照してください.

ここまでは行列の横方向に和をとる, 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 $\mathbf{P}$ を縦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.

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

$$p(Y=\mathtt{c}\,|\,X=\mathtt{a}) \tag{2.17}$$

---

*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.

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

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

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

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

と表すことができます. すなわち一般に, 条件つき確率は $p(X)>0$ のとき

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

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.

2.2 節で計算した同時確率 `p[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:
        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

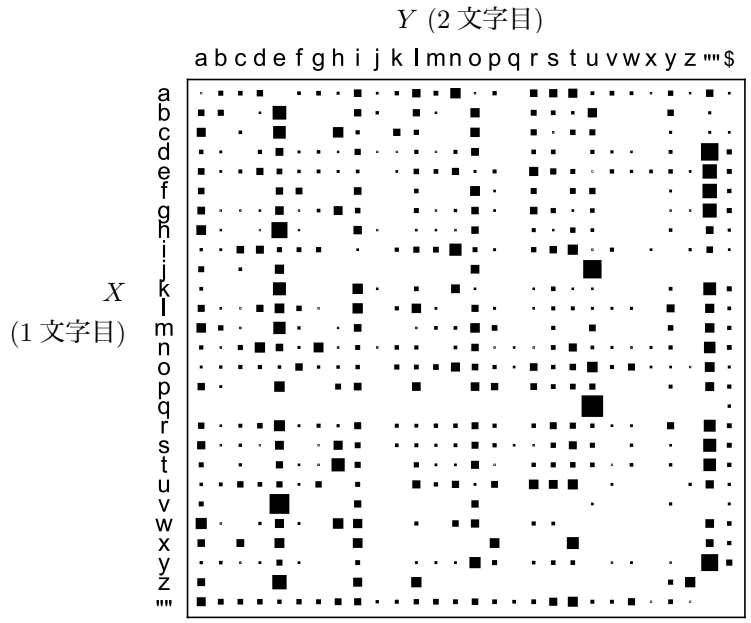


図 2.5: 『不思議の国のアリス』の文字バイグラム条件つき確率 $p(Y|X)$ を表す行列. 確率の総和は各行で 1 になっており, 興味深いパターンがみられることがわかります. 同時確率を表す図 2.3 と比べてみましょう.

$p(X)$, 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. 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とることができるでしょう. (→演習 2–1)

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

*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は限らず, 「両方起こる」という意味です.

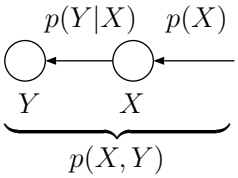


図 2.6: 確率の連鎖則 $p(X,Y)=p(Y\,|\,X)\,p(X)$ のイメージ. X と Y が両方起こるとは, まず X が起こり, その下でさらに Y が起きることと同じです.

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,

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

が明らかに成り立ちます. これを**確率の連鎖則**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 (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) からも簡単に得られます**. 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,

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

が成り立つ場合, つまり式(2.20)と見比べれば

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

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 X の確率は Y の確率に影響し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

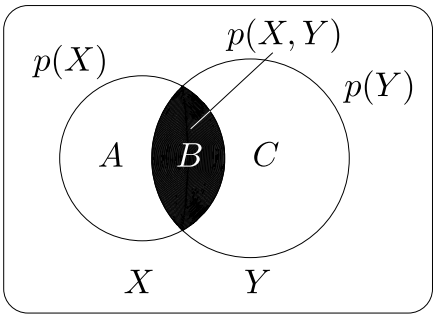


図 2.7: ベン図で表した確率と同時確率, および条件つき確率の関係. 枠で囲った全体の確率を 1 としたとき, 同時確率 $p(X,Y)$ は B の面積になり, 条件つき確率 $p(Y\,|\,X)$ は割合 $B/(A+B)$ のことです. $p(X)$ を表す面積 $A+B$ に条件つき確率 $p(Y\,|\,X)$ をかければ B の面積, すなわち $p(X,Y)$ が得られます.

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 の面積にこの割合をかければよく, $(A+B)\cdot\dfrac{B}{A+B}=B$ 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, $n(\texttt{aa})=1$, $n(\texttt{ab})=214$, $n(\texttt{ac})=157$, $\cdots$, $n(\texttt{a\$})=11$ で, その総和は $n(\texttt{a})=8791$ 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

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

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 の後に任意の文字 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].

確率で書けば, これは, 前に来る文字を x として

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

*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) 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?

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 $p(\mathtt{j}) = 0.0011$, $p(\mathtt{a}) = 0.0662$ 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,
   '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 "oj" 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

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

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

---

*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, 実際の発展や一般化は, 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[24]. とはいえ, ベイズの定理を積極的に利用するベイジアンにとって, ロンドン市内にあるベイズの墓石は今でも聖地となっています.

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 ($X=0$ 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 ($X=1$ 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.

$$\begin{aligned} p(X|Y=0) &= \frac{p(X,Y=0)}{p(Y=0)} = \frac{p(X,Y=0)}{\sum_X p(X,Y=0)} \\ &= \frac{p(Y=0|X)p(X)}{\sum_X p(Y=0|X)p(X)} \end{aligned} \tag{2.32}$$

*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, 本書でもこの後で扱うように, 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.

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,

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

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,

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

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

$$\begin{aligned} p(X|Y=0) &\propto p(Y=0|X)p(X) \\ &= \begin{cases} p(Y=0|X=0)p(X=0) \\ p(Y=0|X=1)p(X=1) \end{cases} = \begin{cases} 1\times 0.2 \\ 0.3\times 0.8 \end{cases} = \begin{cases} 0.2 \\ 0.24 \end{cases} \end{aligned} \tag{2.37}$$

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

$$(\textbf{事後確率}) \propto (\textbf{尤度}) \times (\textbf{事前確率}) \tag{2.39}$$

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 2–12).

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 [*17]. 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の式(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. これを一般化すると, $s = c_1 c_2 \cdots c_T$ のとき, 式(2.20)の確率の連鎖則を次々に用いれば, s の確率は

$$\begin{aligned} p(s) &= p(c_1, c_2, \cdots, c_T) \\ &= p(c_T | c_1, \cdots, c_{T-1})\, p(c_1, \cdots, c_{T-1}) \\ &= p(c_T | c_1, \cdots, c_{T-1})\, p(c_{T-1} | c_1, \cdots, c_{T-2})\, p(c_1, \cdots, c_{T-2}) \\ &= p(c_T | c_1, \cdots, c_{T-1})\, p(c_{T-1} | c_1, \cdots, c_{T-2}) \cdots p(c_2 | c_1)\, p(c_1) \end{aligned} \tag{2.43}$$

---

*17 これは, Shannon による情報理論の最初の論文[25]で行われたものと, 基本的には同じものです.

$$= \prod_{t=1}^{T} p(c_t|c_1,\cdots,c_{t-1}) \qquad \text{(文字列の確率)}$$

と分解することができます. ここで式(2.43)の最後の行で, $t=1$ のときは条件 $c_1,\cdots,c_{t-1}$ は空だと約束します. つまり, 文字列 s の確率は, 確率の連鎖則から, 各文字 c_t の「それまでの文字を知った上での」条件つき確率 $p(c_t|c_1,\cdots,c_{t-1})$ の積として表されるということです. 式(2.43)は常に成り立つ基本的な関係式ですので, 覚えておきましょう.

ただし議論を簡単にするため, 以下では式(2.42)の X,Y,Z で考えてみることにします. このとき文字列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

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

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, $0 \le r < 1$ 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. [*18] 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)
```

とすればよいでしょう.

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[*19] を使って確率を計算しています. 空白␣も文字の一つであることに注意してください.

```
% 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,‘
```

*18 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, この場合のように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[26]. 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[27], 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.

*19 サポートサイトの同じフォルダに置いてあります.

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)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,

$$p(s) = p(X,Y,Z) = p(Z\,|\,Y)p(Y\,|\,X)p(X) \tag{2.45}$$

で与えられます. したがって, 式(2.45)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鍵括弧 “「” 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. [*20] 実装例

---

*20　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

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. *21

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(2.45)式の $p(X)$ は 1 です. したがって, s の確率は

$$p(s) = p(\mathtt{a}|\mathtt{\hat{}})\, p(\mathtt{l}|\mathtt{a})\, p(\mathtt{i}|\mathtt{l})\, p(\mathtt{c}|\mathtt{i})\, p(\mathtt{e}|\mathtt{c})\, p(\mathtt{\$}|\mathtt{e}) \tag{2.46}$$

と, すべてバイグラムで書けることになります. 最初と最後に文頭文字 "^" と文末文字 "$" を導入して文頭および文末との接続確率 $p(\mathtt{a}|\mathtt{\hat{}})$, $p(\mathtt{\$}|\mathtt{e})$ を考えているため, 文字列の「最初が "a" で始まりやすい」「最後が "e" で終わりやすい」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.full.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 ilille sarugl 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.
```

では, 擬似的に多次元配列を実現するための標準の区切り文字として内部で使われています.

*21 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, BOS および 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.

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「隣り同士の文字の確率」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, 文字列がどんなふう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 [*22].

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 [*23]. 図 2.10 に示した, ケンブリッジ大学の機械学習の研究者 MacKay らが開発した Dasher [28] [*24] は, このことを利用した入力システムです. もし病気や障害で指が動かなくても, 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るのです.

これらは文字列に限らず, 3 章の単語列の場合でも同様に成り立ちます [*25].

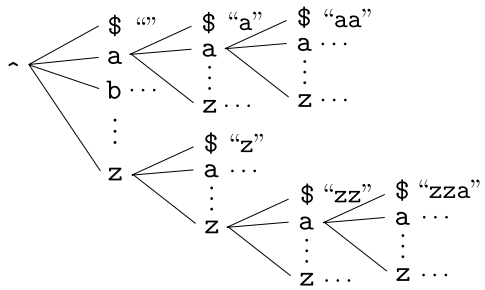


図 2.9: 文字列全体の空間を表す樹形図. ^ で BOS, $ で EOS を表しています.

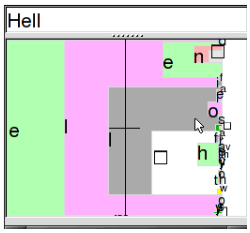


図 2.10: Dasher で “Hello” を入力している様子. ＋印を上下に動かすだけで, 言葉を紡ぐことができます.

*22 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

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

のように条件つき確率を計算するかわりに,

$$p(Z=\mathtt{i}\,|\,X=\mathtt{a}, Y=\mathtt{l}) \tag{2.48}$$

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.[*25] 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

$$p(z\,|\,x,y) = \frac{n(xyz)}{n(xy)} \tag{2.49}$$

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 は予測する任意の文字です.

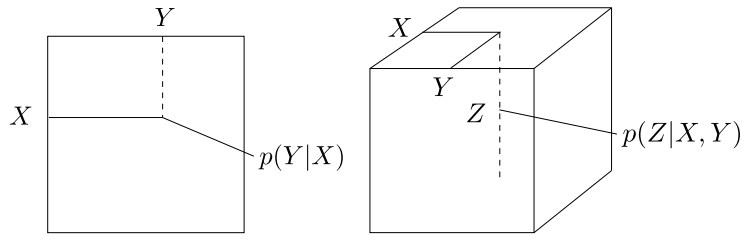
ただし, この場合はユニグラムやバイグラムと異なり, 大きな問題が発生します. 式(2.47)のバイグラムの確率は X と Y の組み合わせの数, つまり文字種の

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[29]によって解析されています.

*23　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ます[30].

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

*25　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. 筆者は文脈ごとに最適な n をベイズ推定することで, 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[31][32].



(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 [*26], 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(2.48)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(2.49)より

$$p(\mathtt{k}|\mathtt{on}) = \frac{n(\mathtt{onk})}{n(\mathtt{on})} = \frac{0}{n(\mathtt{on})} = 0 \tag{2.50}$$

です. そうすると, たとえば文字列 `"monk"` の確率は

*26 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.

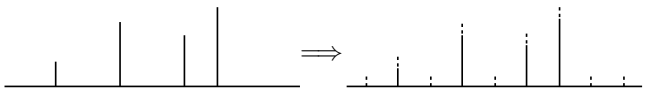


図 2.12: 確率分布の平滑化. 頻度にすべて小さな値 α を足して正規化することで, 離散的な確率分布がより「滑らか」になっています.

$$p(\texttt{"monk"}) = p(\texttt{m}|\texttt{\^{}\^{}}) \cdot p(\texttt{o}|\texttt{\^{}m}) \cdot p(\texttt{n}|\texttt{mo}) \cdot \underbrace{p(\texttt{k}|\texttt{on})}_{=0} \cdot p(\texttt{\$}|\texttt{nk}) = 0 \tag{2.51}$$

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**) といいます.

こうすると, トライグラムの予測確率は式(2.49)に代えて, 次の式で求められます.

$$p(z|x,y) = \frac{n(xyz)+\alpha}{\sum_z (n(xyz)+\alpha)} = \frac{n(xyz)+\alpha}{n(xy)+V\alpha} \tag{2.52}$$

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 [*27].

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**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.[*28] ただし, $\alpha = 0$ にすると平滑化

*27　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 を無限大にと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. 筆者による言語モデル NPYLM [33]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.

*28　トライグラムの場合は, ゼロ頻度問題によって, 式(2.52)でほとんどの文字 z について

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(2.52)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(2.50)の $p($`k`$|$`on`$)$ は式(2.52)より, `alice.full.txt` では $n($`on`$)=1054$, 現れる文字の総数 $V=43$ なので

$$p(\texttt{k}|\texttt{on}) = \frac{0+0.0001}{1054+43\times 0.0001} = 0.00000009 \tag{2.53}$$

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.full.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. [*29]

```
% 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!
```

$n(xyz)=0$ 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.

*29 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. [*30] 図 2.13 に, 中島敦『環礁』`kansho.txt` (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』`ginga.txt` (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

*30　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.

また, 単語は文字からなっていますから, "alice", "fortunately" などの単語をそれぞれ「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 [*31] 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.

---

*31 これはいろいろ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：マルコフモデルと n グラムモデル

47 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 $(n-1)$ 文字だけによる確率モデルです. 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. マルコフモデルという名前は, ロシアの数学者 Andrej Markov (1856–1922) が 1913 年の論文[34]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 文字の中で, 母音と子音が連続する確率を計算したことに端を発しています [*32]. 第一次世界大戦より前のこの時代, まだコンピュータは出現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 [*33] 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(2.52)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 $\alpha = \alpha^*$

*32　この研究は 20,000 文字の統計を単に数えるのではなく, 100 文字の連続× 200 個に分けてそれらの間での統計量のばらつきを求めており, 現代のわれわれにも大変参考になるものです.

*33　もし人手で決める場合も, 49 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, われわれの手元には `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 [*34]

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(2.52)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 [*35],

$$\begin{cases} p(\text{ぎ}) = p(\text{て}) = \dfrac{0+\alpha}{12+87\alpha}\,, \\ p(\text{を}) = p(\text{ま}) = \dfrac{1+\alpha}{12+87\alpha}\,, \\ p(\text{い}) = \dfrac{2+\alpha}{12+87\alpha} \end{cases} \tag{2.54}$$

となります. よって後半の文字列 D' の確率は,

$$p(D' \,|\, D, \alpha) = p(\text{ぎ})\, p(\text{い})\, p(\text{を})\, p(\text{ま})\, p(\text{て}) \tag{2.55}$$

*34 元の句では「柊」だけが漢字ですが, ここでは説明のためひらがなにしています. 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 [35]), 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.

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

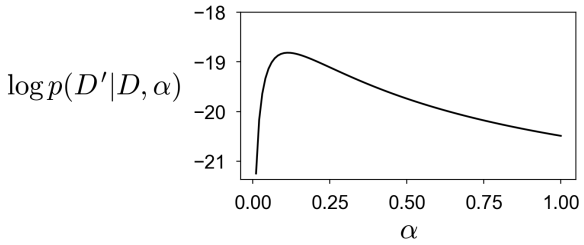


図 2.15: 俳句の仮想例での評価データ D' の予測確率の対数 $\log p(D' \mid D, \alpha)$ とパラメータ α. この場合, $\alpha = 0.114$ で D' の確率が最大になっています.

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

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(2.55)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 [*36], $\alpha^* = 1/96(\sqrt{5761}-65) \approx 0.114$ 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(2.52)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, 基本的な方法は同じです. 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**あるいは**開発データ**, 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といいます[36]. 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

---

*36　*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.

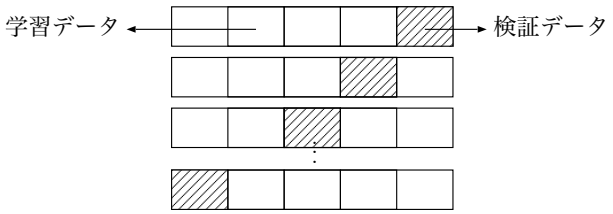


図 2.16: K 分割クロスバリデーションの様子. データを K 個 (ここでは $K=5$) に等分し, そのうち 1 つを検証データ, 残りの $K-1$ 個を学習データとする計算を K 通り行い, 結果を平均して最終的な答えとします.

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 $1, \cdots, N$ をランダムに並び替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 [*37] 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,

(2.56)

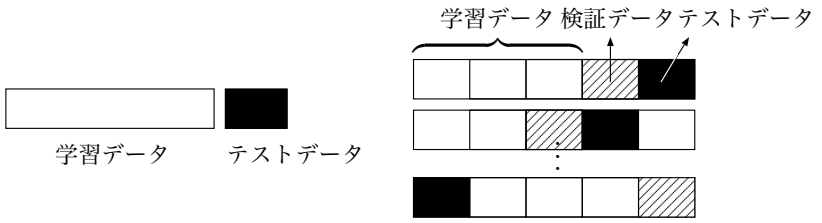
$$\log p(D_1 \,|\, D_2, D_3, D_4, D_5) + \log p(D_2 \,|\, D_1, D_3, D_4, D_5) + \cdots + \log p(D_5 \,|\, D_1, D_2, D_3, D_4)$$

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, 2.6.3 節で説明する 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)
```

---

*37　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 [*38]. 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(2.56)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**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. しかし, 実は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, 厳密にいうと, これは一種のチートに

*38 ベイズ最適化[37]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.

なっているからです. 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**といいます. 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.[*39]

2.6.2　テキストの確率の計算

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. テストデータとして, テキスト D が文 (本章では文字列) の集合

$$D = \{s_1, s_2, \cdots, s_N\} \tag{2.57}$$

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

*39　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.

$$p(D) = \prod_{n=1}^{N} p(s_n) \tag{2.58}$$

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(2.43)で示したように

$$p(s_n) = \prod_{t=1}^{T} p(c_t \,|\, c_1, \cdots, c_{t-1}) \tag{2.59}$$

として計算できます.

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(2.58)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(2.51)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

$$p(s) = \left(\frac{1}{10}\right)^{20} = 0.00000000000000000001 \tag{2.60}$$

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

$$\log p(s) = 20 \log \frac{1}{10} = -46.05 \tag{2.61}$$

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(2.61)を文字列の長さ T で割って

$$\frac{1}{T} \log p(s) = \frac{1}{T} \sum_{t=1}^{T} \log p(c_t \,|\, c_1, \cdots, c_{t-1}) \tag{2.62}$$

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,

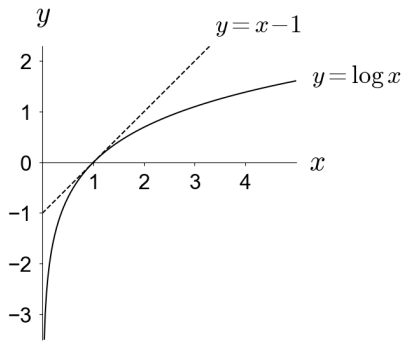


図 2.18: 対数関数 $y = \log x$ のグラフ. 点線で表したのは $x = 1$ での接線 $y = x - 1$ で, $\log x$ はつねにこの接線の下にあり, $\log x \leq x - 1$ が成り立っています.

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

が 1 文字あたりの確率の対数になります. なお, 式(2.62)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(2.62)で用いた, 確率の対数

$$\log p(c_t \,|\, c_1, \cdots, c_{t-1}) \tag{2.63}$$

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.

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.

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[*40] の**自己情報量**といい,

$$I(x) = \log \frac{1}{p(x)} = -\log p(x) \qquad (\textbf{自己情報量}) \tag{2.64}$$

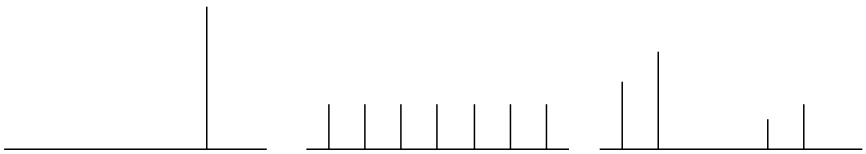
と書きます. 対数の底は任意ですが, 上のように 0/1 の決定をもとに底を 2 にする場合は bit (binary digit の略), 自然科学や計算機上の対数で通常用いられている e を底にする場合は nat (natural digit) が単位となります. [*41] たとえば, 確率 $p(x) = 0.07$ の単語が出現したとき, これは $1/0.07 = 14.3$ 個の等価な選択肢から 1 つが選ばれたことと同じなので, その情報量は

$$I(x) = \log \frac{1}{0.07} = \log 14.3 = 2.66 \text{ (nat)} \tag{2.65}$$

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.

∗40 Claude Shannon (1916–2001) は情報理論を創始したアメリカの通信工学者・数学者です. 本書では説明しませんが, 情報理論における雑音あり通信路[38]の考え方は, 自然言語処理においても大変有用なモデルとなっています.

∗41 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 $H = -\sum_x p(x) \log p(x)$.

メモ：符号長について

底となる数を ℓ としたとき, 情報量とは対応する分岐数を ℓ 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(2.64)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ます. 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[38, 30].

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(2.64)で表される x の自己情報量 $I(x)$ を, $p(x)$ で期待値をとった

$$H = \sum_x p(x) I(x) = -\sum_x p(x) \log p(x) \tag{2.66}$$

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,

$$H = -\sum_x p(x) \log p(x) = \big\langle -\log p(x) \big\rangle_{p(x)} \tag{2.67}$$

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. たとえば, アルファベットが $\mathcal{X} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ の 4 つで, 確率分布が $(p(\mathtt{a}), p(\mathtt{b}), p(\mathtt{d}), p(\mathtt{d})) = (0.4, 0.2, 0.1, 0.3)$ のとき, そのエントロピーは

$$H = -\,0.4\log 0.4 - 0.2\log 0.2 - 0.1\log 0.1 - 0.3\log 0.3 = 1.28$$

になります. (→演習 2–13)

図 2.19 に示したように, $p(x)$ が $(0, 1, 0, \cdots, 0)$ 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(2.66)から $H = 1 \cdot (-\log 1) = 0$ です.[*42] 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. 符号長の言葉でいえば, 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(2.62)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

$$\frac{1}{T}\log p(s) = \frac{1}{T}\sum_{t=1}^{T}\log p(c_t\,|\,c_1, \cdots, c_{t-1}) \tag{2.68}$$

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 0.7 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

*42 なお, $0\log 0 = 0$ と約束します.

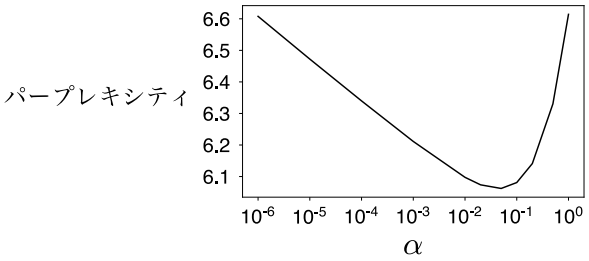


図 2.20: テストデータ `alice.test.txt` のパープレキシティと, 文字トライグラム言語モデルの α の値の関係. 横軸は対数軸になっていることに注意してください.

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.[*43] すなわち, パープレキシティとは**平均分岐数**のことです. たとえば, いまの例では, パープレキシティが $1/0.01=100$ 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(2.62)から, 次のようにして計算します.

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

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 $1/(1/V)=V$ 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.[*44] 図 2.20 に, 異なる α の値について `alice.txt` の

*43　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.

*44　パープレキシティは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欠点

テストデー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 (本章では 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分布 $p(x)$ 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. [*45]

$$D(p||q) = \sum_{x \in \mathcal{L}} p(x) \log \frac{p(x)}{q(x)} \tag{2.70}$$

KL ダイバージェンス $D(p||q)$[*46] 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

$$D(p||q) \geq 0 \qquad (\textbf{KL ダイバージェンスの非負性}) \tag{2.71}$$

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,

---

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[39]. PPLu では, 式(2.69)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, 3.3 節の議論から文脈 $c_1, \cdots, c_{t-1}$ 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[39]を参照してください.

*45 Solomon Kullback (1907–1994) と Richard Leibler (1914–2003) はアメリカの暗号研究者・数学者で, Kullback は第二次世界大戦中に日本軍の暗号を解くのに大きな役割を果たしました. KL ダイバージェンスは, 1951 年の論文[40]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.

*46 KL($p||q$) と書くこともあります.

$$\sum_x p(x)\log\frac{q(x)}{p(x)} \le \sum_x p(x)\left(\frac{q(x)}{p(x)}-1\right) = \sum_x q(x) - 1 = 0 \tag{2.72}$$

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

$$\sum_x p(x)\log\frac{p(x)}{q(x)} \ge 0 \tag{2.73}$$

が得られます. □ [*47]

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.[*48] 実際, 式(2.70)を書き直すと

$$\begin{aligned}\sum_{x\in\mathcal{L}} p(x)\log\frac{p(x)}{q(x)} &= \left(\sum_{x\in\mathcal{L}} p(x)(-\log q(x))\right) - \left(\sum_{x\in\mathcal{L}} p(x)(-\log p(x))\right) \\ &= \big\langle -\log q(x)\big\rangle_{p(x)} - \big\langle -\log p(x)\big\rangle_{p(x)}\end{aligned} \tag{2.74}$$

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「真の分布 p で符号化したときの符号長」の期待値の差を計算していることになります. [*49] 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, 真の分布 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

$$D(p||q) = \sum_{x\in\mathcal{L}} p(x)\log\frac{p(x)}{q(x)} \tag{2.75}$$

---

*47　この □ は, 一般に数学で「証明終わり」を意味する記号です.

*48　このため, $D(p||q)$ は距離の公理を満たさず, 厳密には数学的な「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, 本章末で紹介する情報理論の教科書を参照してください.

*49　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.

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

です. これから,

$$-\sum_{x\in\mathcal{L}} p(x)\log q(x) \geq -\sum_{x\in\mathcal{L}} p(x)\log p(x) = H(p) \tag{2.76}$$

が成り立つことがわかります. 式(2.76)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**とよばれる量

$$H(p,q) = -\sum_{x\in\mathcal{L}} p(x)\log q(x) \qquad (\textbf{クロスエントロピー}) \tag{2.77}$$

です. これは言語のあらゆる文 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 $p(x)$ による期待値になっています. すなわち式(2.76)は, あらゆる確率分布 p, q について, q の p に関するクロスエントロピーは p のエントロピーより大きく,

$$H(p,q) \geq H(p) \qquad (\textbf{クロスエントロピーとエントロピー}) \tag{2.78}$$

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(2.75)より, **クロスエントロピーの最小化は, 真の確率分布と統計モデルの間の 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(2.76)および式(2.78)の左辺のクロスエントロピーは

$$-\sum_{x\in L} p(x)\log q(x) \simeq -\frac{1}{N}\sum_{n=1}^{N}\log q(x_n) \tag{2.79}$$

とモンテカルロ近似することができます.[*50] 式(2.79)を式(2.76)の左辺に代入

*50 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, $\int p(x)f(x)dx \simeq$

して, 両辺を指数の肩にのせれば, [*51]

$$\mathrm{PPL} = \exp\left(-\frac{1}{N}\sum_{n=1}^{N}\log q(x_n)\right) \geq e^{H(p)} \tag{2.80}$$

が得られます. 式(2.80)の左辺は式(2.69)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(2.80)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, 元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

$\frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})$ 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[41]や, 教科書[30]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[42].

*51　$y = e^x$ 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[25] 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[*52], 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 [*53] 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 [43]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. 80)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. [*54]

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, よい統計モデル $q(D)$ というわけです.

---

*52 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.

*53 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.

*54 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.

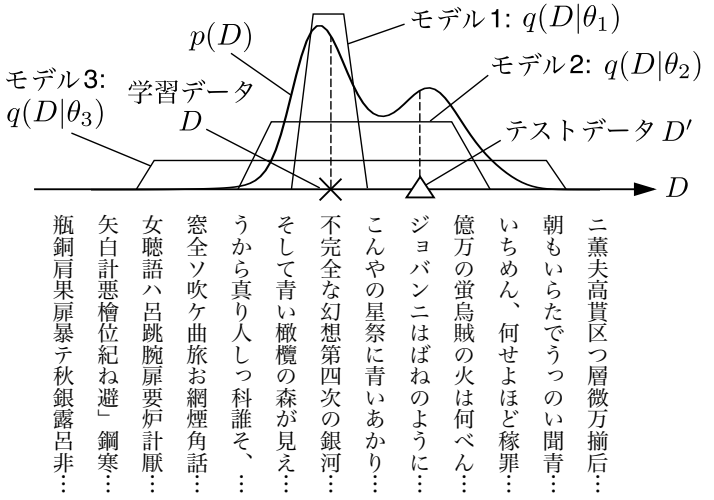


図 2.21: 統計モデルの過学習と汎化性能. 横軸は可能なデータの空間を, 縦軸はモデルの与える確率を表しています. モデル 1 は与えられた学習データ D (×印) に過学習してしまい, テストデータ D' (△印) に非常に低い確率しか与えられなくなっています. モデル 3 は一般的すぎて逆に過少適合となるため, テストデータに一番高い確率を与えるのは, 真のデータ分布 $p(D)$ に近く, 汎化性能が高いモデル 2 となっています. 図の下に, 『銀河鉄道の夜』を学習データとした場合の可能なデータ空間の例を示しました.

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 (×印) 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' (△印) 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**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.

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および情報理論の基礎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.

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. 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.

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. [*55] 現代の深層学習も, すべてこうした統計的な概念をもとに開発・改良されています.

---

*55 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 [44]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」などの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`(オーク) や`sed`(セド)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.[*56] 51 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, Excel で行うような計算をコマンドラインから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` [*57] 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, 入力テキストの 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.

*56 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.
*57 `ed` については, 『The UNIX Super Text』[45]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 [*58] やチェス [*59] すら存在します.

sed や awk については『sed&awk プログラミング』[46]『プログラミング言語 AWK』[47] といった標準的な参考書があるほか, 「awk の簡単な使い方」[*60]「SED 教室」[*61]「sed は日暮れて」[*62] 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[*63] [48]やレイトレーシングなど, 驚くべきプログラムも存在しています.

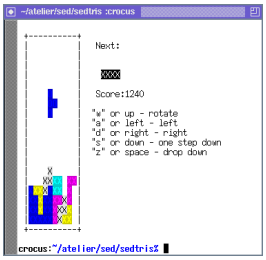


図 2.22: 文字端末で, sed で書かれたテトリスをプレイしている様子.

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

2 章の演習問題

[2–1] 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.

[2–2] 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.

[2–3] 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.

[2–4] 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 [*64]. 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.

[2–5] 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.

[2–6] 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.

---

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

*64　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.

[2–7] 本章で紹介した言語モデルよりも性能のよい, 3.4.3 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.

[2–8] 英語や日本語の適当なテキストを学習データとテストデータに分割し, 学習データで推定した文字 n グラムモデルを用いて, テストデータのパープレキシティを計算してみましょう. n を変えたり, 式(2.52)の加算平滑化の係数 α を変えると, パープレキシティはどう変わるでしょうか.

[2–9] 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, 51 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.

[2–10] 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, 51 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.

[2–11] 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 [*65]. 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.

---

*65 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.

[2–12] 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]
```

[2–13] 確率分布を

```
0.2 0.4 0.05 0.25 0.1
```

のように空白や改行で区切られたテキストファイルとして与えると, そのエントロピー H を出力するプログラムを書いてみましょう. ディリクレ分布 (3 章) からランダムに生成した確率分布のエントロピーを計算すると, どうなるでしょうか. ディリクレ分布のパラメータ $\boldsymbol{\alpha}$ によって, エントロピーはどう異なるでしょうか.

2章の文献案内

本章では文字の言語モデルを題材に, 確率と統計モデルの基礎について説明しました. ここで扱った統計モデルの基本的な話題は, 2022年に出版されたMurphyによる“Probabilistic Machine Learning: An Introduction” (PML) [49] *66 のような機械学習の教科書で基礎から順を追って説明されており, わかりやすいでしょう. 日本語版の刊行も予定されています. 統計モデルの数学的な計算は, Bishopによる2006年の“Pattern Recognition and Machine Learning” (通称PRML) [36] *67 で詳しく解説されています. 筆者も翻訳に参加した日本語版[50]もあり, 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” [30] は, ベイズ統計と情報理論に基づく深い考察に支えられた機械学習の名著で, こちらも全文のPDFが公開されています *68. 本章前半の議論は, 多くこの本を参考にしました. ベイズの定理の使い方に慣れるには, [30]の短い3章“More about Inference”を読まれることを特にお勧めします.

自然言語処理に現れる機械学習については, 高村による『言語処理のための機械学習入門』[21]がよく読まれています. 本書と異なりベイズ的な説明はない一方で, 分類問題に関する記述は充実していますので, 教師あり学習でSVMやCRFといった分類器を使う場合は, こちらを参考にするとよいでしょう.

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. ある意味で, `0/1` や `a..z` のアルファベットの系列を主に扱う情報理論を, 単語や文, 文書といった上位の構造を含めて徹底的に高度化したのが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通信に関わる, 多くの技術的内容が含まれています. 情報理論では, CoverとThomasの教科書[42] (和訳[51]) が最も基本的な文献として知られています. また, MacKayによる[30] でも詳しく扱われています.

---

*66 `https://probml.github.io/pml-book/book1.html` で, 全文の草稿PDFが公開されています.

*67 PRMLも, `https://www.microsoft.com/en-us/research/people/cmbishop/prml-book/` で全文のPDFが公開されています.

*68 `http://www.inference.org.uk/mackay/itila/book.html`

日本語では, 韓と小林による『情報と符号化の数理』[38] 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, 国際会議 COLING 2022 のチュートリアル “Information Theory in Linguistics: Methods and Applications”[*69] 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” [52] [*70] 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, かつ奥深い論考です.

人文学の分野でも, 言語の数学的な取り扱いは一部で行われてきました. 計量国語学者の水谷静夫による『言語と数学』[53]『数理言語学』[54]はそれぞれ 1970 年代と 1980 年代の出版ながら, 非常にレベルが高く, 現代でも示唆に富むものです. 筆者は学部生のときに大学の図書館の棚で『言語と数学』に出会い, 言語を数学的に扱えることに感銘を受けました. テキストデータの使用も以前から行われており, 2001 年の近藤らの論文[55]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. 『人文科学の FORTRAN 77』[56] は東大での講義のための教科書で, 50 年近く前の本でありながら, 「人文向け」のイメージと異なって例題や問題が非常に興味深いものです. 言語が Fortran という問題は置いておいても, 人文系の方にぜひ推薦できる教科書です. 人文学におけるデータ解析の最近の教科書としては, Python を用いた実例を示した “Humanities Data Analysis” [57]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利用については, 5 章の文献案内をご覧ください.

∗69　https://rycolab.io/classes/info-theory-tutorial/
∗70　https://www.ism.ac.jp/~iba/a19.pdf

3　単語の統計モデル

3.1　文字から単語へ

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

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

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

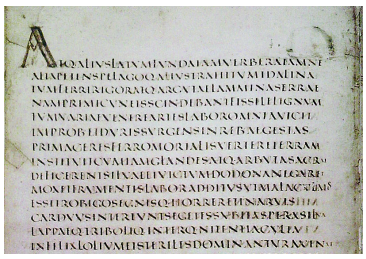


図 3.1: ヴェルギリウスのラテン語のテキスト (西暦 141 年ごろ) での, 単語の続け書きの例. *1

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

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

うなすべての慣用句を単語として認めるかが決まっているわけではありません. したがって, 何が「単語」であるかという基準には本質的に曖昧性があり, 唯一の正解があるわけではありませんが, 多くの言語は現在では空白で区切って書かれますし, 日本語や中国語も, 最適とはいえなくても, 下に述べるように標準的な方法で単語に区切ることができますので, 本書でもそれを使っていくことにします. 上のような慣用句を統計的に自動認識する方法については, この後の 3.3 節を参照してください.

文字列を単語に分解することは**単語分割**, あるいは動詞や名詞といった品詞の付与や活用形の認識も行う場合は**形態素解析**とよばれています. たとえば, 標準的な形態素解析器の一つである MeCab[*2] を使えば, 日本語の文字列を簡単に単語に区切ることができます. Python の場合, これにはまず, `MeCab` モジュールをインストールします. Google Colaboratory の Python 環境を使っている場合は, 執筆時では次のようにすればインストールすることができます (>は Colaboratory のプロンプト).

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

この上で, 下のようにすれば文字列が単語に分割されます.

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

したがって, たとえば『銀河鉄道の夜』のテキスト `ginga.txt` は次のようにして単語に分解することができます.

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

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

⇒ [’銀河’, ’鉄道’, ’の’, ’夜’]
[’一’, ’、’, ’午后’, ’の’, ’授業’]
[’「’, ’では’, ’みなさん’, ’は’, ’、’, ’そういう’, ’ふう’, ’に’, ’川’, ’だ’, ’と’, ’云わ’, ’れ’, ’たり’, ...

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

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

なお, 形態素解析器にはほかにも kuromoji, GiNZA, KyTea, …など様々なものがあります[*3] ので, 使いやすいものを用いるとよいでしょう.

ノート：教師なし形態素解析

何が「単語」なのかを人間が決めるには限界があるため, 筆者は, あらゆる言語の文字列だけから直接,「単語」を教師なし学習する**教師なし形態素解析**の研究を行いました[33, 58]. 教師なし形態素解析では, 観測値である文字列の確率を最大にする分割が単語であると考え, 文字列の単語分割を MCMC 法 (5 章) で次々と学習します. たとえば, 文字列 “今日見た花” は, “今日/見/た/花” と分割する方が, “今/日見/た花” と分割するよりも確率が高いでしょう. ここで確率が高いとは, 単語自身の綴りや, その前後の n グラムの繋がりがデータ全体から見たときに自然だということです. 内部では, 単語の長さが文字種ごとの混合ポアソン分布に従うとして補正する統計モデルも組み込まれています.

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

この方法を用いると, `alice.full.txt` から空白をすべて除いた文字列を, まったく辞書を使わずに図 3.2 のように分割することができます.

```
lastly,shepicturedtoherselfhowthissamelittlesisterofherswould,in
theafter-time,beherselfagrownwoman;andhowshewouldkeep,thro
ughallherriperyears,thesimpleandlovingheartofherchildhood:and
howshewouldgatheraboutherotherlittlechildren,andmaketheireyes ...
                                ⇓
last ly , she pictured to herself how this same little sister of her
s would , inthe after - time , be herself agrown woman ; and
how she would keep , through allher ripery ears , the simple
and loving heart of her child hood : and how she would gather
about her other little children ,and make theireyes ...
```

図 3.2:『不思議の国のアリス』の教師なし形態素解析の結果 (一部).

このデータは 115,961 文字と非常に小さいため, 一部の単語は結合されていますが, 英語の「単語」が非常によく復元できることがわかります. この方法は辞書や教師データを一切必要としないため, 古文や方言, 未知の言語にも適用することができます. 図 3.3 に,『源氏物語』の全文で学習した場合の冒頭部を示しました [*4].

教師なし形態素解析は, 3.4.3 節で説明する Kneser–Ney 平滑化の理論モデルである階層 Pitman–Yor 過程による単語 n グラムモデル[59]において, 単語を生成する基底測度に文字の ∞ グラム言語モデルが埋め込まれている, 高度な階層ベイズモデルです. 4 章で説明する Gibbs サンプリングと動的計画法を用いて学習されます. 本書のレベルを大きく超えるため詳細は割愛しますが, C++と Python による再現実装が公開されています [*5].

```
いづれ の 御 時 に か 、 女御 更衣 あまた さぶらひ たまひ け る
中に 、 いと やむごと なき 際 に は あら ぬ が 、 すぐれて 時め
き たまふ あり けり 。 はじめより 我 は と 思ひ あが り たまへ
る 御方々 、 めざまし き ものに おとしめ そねみ たまふ 。 同
じ ほど 、 それ より 下 臈 の 更衣 たち は 、 まして やす から …
```

図 3.3:『源氏物語』の教師なし形態素解析の結果 (一部).

---

*4　『源氏物語』はひらがなの連続が多く複雑なため, この MCMC 法には一週間程度の計算を必要としました.

*5　`https://github.com/musyoku/python-npylm`　筆者が NTT 研究所在籍時の研究のため,

3.2 単語の統計と巾乗則

さて, こうしてテキストが単語に分けられたとき, 文字のモデルと最も異なっている点は何でしょうか. 最も重要なのは, 「単語の種類は無限にある」ということでしょう. それぞれの言語で使われる文字の種類は有限ですが, 文字の組み合わせである単語は, 長さに通常は制限がありませんので, いくらでも多くの単語を作り出すことができます. *6

とはいえ, 計算機上で無限の語彙を表すのは難しいため *7, ほとんどの場合は, 一定の基準で語彙を選ぶことになります. 実際に, われわれが普段用いている辞書はこうして一定の語彙を選んだものです. 表 3.1 に, 日本語および英語の主な辞書に掲載されている語彙 (見出し語の数) の例をまとめました. これを見ると, 単語として必要な語彙の種類は文字の種類よりはるかに大きく, 最低でも数万語, 多い場合には数十万語を超えることがわかります. つまり, 統計モデルとして見ると, 単語を考える統計モデルは出力が数万次元を超える**超高次元の統計モデル**になるということです. これが, 文字の統計モデルとの大きな違いです.

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

```
from collections import defaultdict
```

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

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

原実装 (C++で 7000 行程度) は非公開ですが, 細かい点でやや異なるところもあります.

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

*7 筆者による教師なし形態素解析のための言語モデル NPYLM [33]は, 文字 ∞ グラムと単語 n グラムを組み合わせることで, 無限の語彙を扱うことが可能です.

```
freq = defaultdict (int)
with open ('ginga.words.txt', 'r') as fh:
    for line in fh:
        words = line.rstrip('\n').split()
        for word in words:
            freq[word] += 1
for w,c in freq.items():
    print ('%s -> %d' % (w,c))
⇒ 銀河 -> 25
鉄道 -> 4
の -> 1266
夜 -> 6
一 -> 45
、 -> 988
午后 -> 2
授業 -> 2
「 -> 293
では -> 10
みなさん -> 4
...
len(freq)
⇒ 2594
```

『銀河鉄道の夜』では語彙の大きさは 2594 であり, 各単語の頻度は上のようになっていることがわかります. 他のテキストについても同様に数えた語彙数を, 表 3.2 に示しました. 『銀河鉄道の夜』や『不思議の国のアリス』といった (計算機にとっては) 短いテキストでは語彙は数千語程度ですが, 一般的な中規模のテキストでは数万語以上, 大きなテキストでは数十万語や数百万語を超えることがわかります. *8 なお, このように一般的に共有されているテキストデータおよび (あれば) 付随データのことを, **コーパス**ともいいます.

コーパスにはさまざまなものがありますが, 1967 年に編纂された Brown コーパス[62]は分野が偏らないようにサンプリングされた (これを**均衡コーパス**といいます), 最初の大きなコーパスです. アメリカ英語約 100 万語のテキストとその品詞からなっており, 現在では NLTK のようなツールキットにも付属して公開されています *9. 現代の均衡コーパスとしては, 英国では 1 億語の British

*8 Web から取得した 1 兆語の英語テキストから求めた Google 1T 5-gram データ[60]では語彙は 1350 万語, 日本語 2500 億語の 7-gram データ[61]では 256 万語にものぼります.

*9 http://www.nltk.org/nltk_data/

National Corpus (BNC)[*10], 米国では 1500 万語の American National Corpus (ANC)[*11] があり, いずれもデータをフリーでダウンロードできます. 日本語では, 国立国語研究所が約 1 億語の現代日本語書き言葉均衡コーパス (BCCWJ)[*12] を公開しており, オンライン検索は無料で, オフラインのデータは有料で使用することができます. また有料ではありますが, 毎日新聞, 読売新聞, 朝日新聞といった新聞のテキストも近年のものはすべて電子化されており [*13] 購入すればコーパスとして使用することができます. 多言語のコーパスとしては, ドイツで公開されている Leipzig コーパス[63]には 100 言語以上の文が, それぞれ 1 万文から 100 万文含まれており (日本語も Web とニュースの文がそれぞれ 100 万文あります), フリーでダウンロードすることができます [*14].

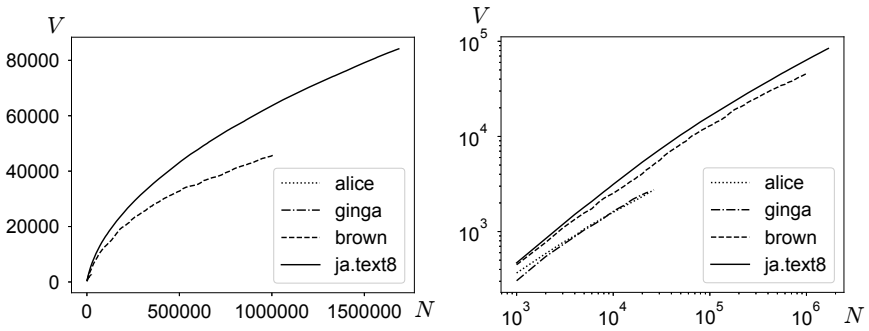
また, 3.5 節で説明する単語ベクトルの学習などを手軽に試すためのコーパスとして, Wikipedia からランダムに 100MB 分のテキストを抽出した text8[*15], および日本語版の ja.text8[*16] はフリーのため, 研究や開発目的で広く使われています. サポートページに, Brown コーパスのテキストを `brown.txt`, text8 および日本語 text8 を `text8`, `ja.text8` として置いておきました. 元々の text8 には改行がないため, 改行を復元したものはそれぞれ `text8.txt`, `ja.text8.txt` となっています.

3.2.1 Heaps の法則

単語の種類に制限はありませんから, 語彙は一般に, テキストが長くなるほど増えていきます. 以下では, テキストの長さは単語数で数えることにしましょう. 先ほどの実行例で, テキストを読みながら頻度辞書 `freq` の大きさ `len(freq)` を見れば, 語彙がどのように増えるのかを確かめることができます. 図 3.4 に, こうして調べたテキストの長さ N と語彙 V の関係を示しました. テキストが

---

*10 `http://www.natcorp.ox.ac.uk/`
*11 `https://anc.org/`
*12 Balanced Corpus of Contemporary Written Japanese の略で, `https://clrd.ninjal.ac.jp/bccwj/` から使用・入手することができます.
*13 `https://www.nichigai.co.jp/dcs/index5.html`
*14 `https://wortschatz.uni-leipzig.de/en/download`
*15 `https://mattmahoney.net/dc/textdata.html`
*16 `https://github.com/Hironsan/ja.text8`



(a) 線形スケールの場合　(b) 両対数スケールの場合

図 3.4: テキストの長さ N と語彙の大きさ V の関係 (Heaps の法則). `alice`：『不思議の国のアリス』, `ginga`：『銀河鉄道の夜』, `brown`： Brown コーパス, `ja.text8`：日本語 text8 コーパスをそれぞれ表しています. `alice`, `ginga` は短いため, 線形スケールではほとんど見えなくなっています.

長くなるほど, 語彙は増えてきれいな曲線を描き, 対数スケールで見るとテキストの言語やジャンルによって多少の違いはあるものの, おおむね同じように語彙が直線的に伸びていくことがわかります. このことは, **Heaps の法則**とよばれています.[*17] Heaps の法則は, 次の式で表されます.

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

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

テキスト	長さ	語彙	頻度≥2	頻度≥10
`alice.txt`	26,396	2,748	1,471	379
`brown.txt`	1,012,603	46,055	26,481	8,395
日本語 text8	15,160,499	249,629	126,690	45,024
毎日新聞 2011 年度	21,805,730	129,971	88,669	42,465
New York Times 2008 年度	65,333,240	249,610	158,735	72,554

∗17　Heaps の法則の名前は, 情報検索の分野で Harold Stanley Heaps が 1978 年に出版した本[64]によるものですが, これより前に, 初期の計量言語学者である Gustav Herdan (1897–1968) が 1960 年にこの法則を発見していました[65]. よって, これを Herdan-Heaps の法則ということもあります.

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

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

になりますから, $\log V$ と $\log N$ は傾き γ の比例関係となります. これが, 両対数スケールの図 3.4(b) で直線関係が現れている理由です.

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

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

が Zipf の法則ですから, 確率分布にするための正規化定数は

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

です. この Z は $\alpha > 1$ のとき, 一定の値に収束することが知られています.[*18] よって Zipf の法則の下では, 頻度順で r 番目の単語の確率は

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

と表されるわけです.

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

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

が成り立つはずです. 両辺の対数をとって整理すれば,

---

*18 数学では, リーマンのゼータ関数 $\zeta(\alpha)$ と呼ばれています. よく知られているように $\zeta(2) = \sum_{r=1}^{\infty} 1/r^2 = \pi^2/6 \simeq 1.645$ で, 一般に $\zeta(\alpha) < 1 + 1/(\alpha - 1)$ となります.

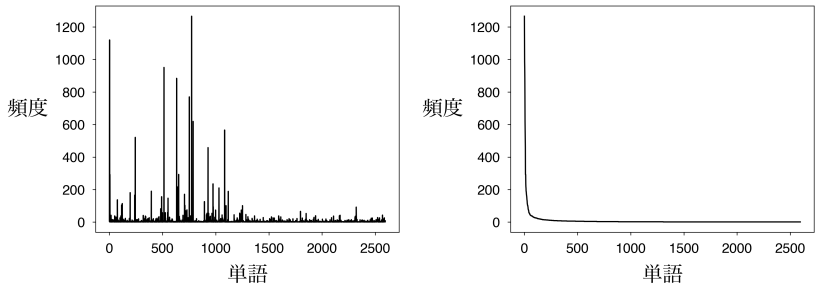
(3.7)
$$-\alpha \log V = \log Z - \log N$$
$$\therefore \quad V = Z^{-1/\alpha} \cdot N^{1/\alpha}$$

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

3.2.2　Zipf の法則

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

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



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

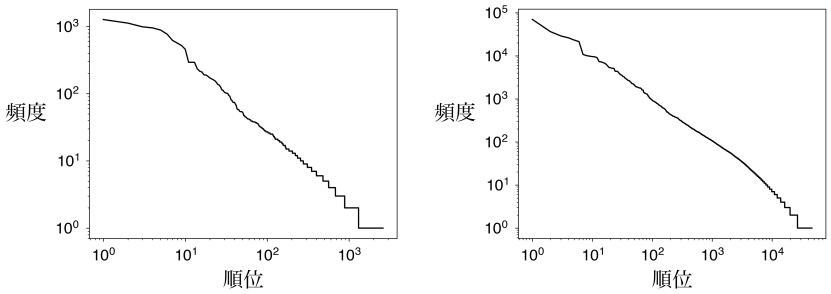
図 3.5: 『銀河鉄道の夜』における単語の頻度.

この結果からわかる通り, 句読点を除くと「の」「に」「を」といった一部の単語に頻度が集中しており, 一方でテキストに一度しか出現していないような「函」「橄欖」といった言葉が大量にあることがわかります.[*19] これらは 1 回しか出現していないために情報が少なく, 統計的な分析からは除いた方がよいでしょう. 一般に, テキスト全体を通した頻度がたとえば 10 以上 (大きいテキストならば 100 以上) のように, 基準を決めて語彙を選択することがよく行われます [*20]. 表 3.2 の最後の列に, 頻度 10 以上の単語を用いた場合の語彙の大きさを示しました. ただし今は, そうした語彙の選択は行わないことにします.

それでは, 上で求めた単語の頻度は, 全体としてはどのように分布しているのでしょうか. 図 3.5(a) に, 『銀河鉄道の夜』に現れる 2594 種類の単語を横軸に辞書順にとり, 縦軸にその頻度をとったプロットを示しました. 先にみたように, 日本語では「の」「に」「を」といった一部の単語が高い頻度を持っていますので, ヒストグラムにところどころ, 突出した値があることがわかります. 横軸の単語を頻度順で並び替えると, 図 3.5(b) のようになります. 確かに, 非常に少数

∗19 こうした一度しか出現しなかった語のことを, コーパス言語学では孤語あるいは hapax legomenon (ギリシャ語で「一度だけ書かれた」の意味) といいます. これは, あくまで与えられたテキストについての概念であることに注意してください. たとえば, 「放課後」はこのテキストでは孤語ですが, 一般にはごくありふれた単語です.

∗20 一般にコーパスには, たとえば新聞記事のように複数のテキストが含まれていますので, 頻度に基づいて語彙を選択することと, あるテキストの中でその語彙に含まれる単語が 1 章で説明したように低頻度で現れることは, 別の問題です.



(a) 『銀河鉄道の夜』の場合. 図 3.5(b) を両軸について対数でプロットしたもの.　(b) Brown コーパスで同様に計算した場合.

図 3.6: 単語の頻度順位と出現頻度の両対数プロット. 順位と頻度が反比例する Zipf の法則が現れています.

の単語に頻度が集中しているのを見てとることができますね.

ただ, 頻度があまりに一部の単語に集中しているため, 様子を詳しく見るために, 縦軸を頻度 f の対数すなわち $\log f$ で, 横軸も同様に頻度の順位 r の対数 $\log r$ で両対数プロットにしてみましょう. こうすると, 図 3.5(b) は図 3.6(a) のように表されます. この図をよく見ると, $\log f$ と $\log r$ の間には**傾きがほぼ** -1 **の比例関係がある**ことがわかります. すなわち, ほぼ

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

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

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

あるいは,

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

という関係が成り立つことになります. すなわち, 単語の相対的な頻度は頻度順に順位が $r = 1, 2, 3, 4 \cdots$ と下がるほど, $1, 1/2, 1/3, 1/4, \cdots$ と減っていく, ということです. この関係は, 他のどんなテキストに対してもほぼ成り立つことが知られています. 図 3.6(b) に, Brown コーパス `brown.txt` に対して同様に計算

したプロットを示しました. ここでも, 同じ法則が成り立っていることがわかります. この法則は, 1930 年代にこの法則を体系化したハーバード大学の言語学者Zipf(ジップ) [*21] の名前をとって, **Zipf の法則**とよばれています. Zipf の法則は, 言語に限らず社会全般で広く成り立つ**巾乗(べき)則**として認知されており, たとえば都市の大きさ, 収入の分布, 地震の規模と頻度など多くの実際の現象が巾乗則に従うことが知られています[69]. 言語を中心としたこうした巾乗則については, [70]で詳しく論じられていますので参照してください.

この Zipf の法則によれば, 語彙が 10,000 個であれば, 最もよく現れる単語の確率と最も現れにくい単語の確率の比は, 1:1/10000 で 10,000 倍にもなる, ということになります. 文字の場合は, 2.1 節で見たように英語では最も頻度の高い文字 'e' と最も低い文字 'z' の確率の比は 150 倍くらいでしたから [*22], 語彙の多い単語の場合は, その差は圧倒的に大きくなっていることがわかります.

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

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

です. よって, その総和は

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

となります. しかし, 式(3.4)でみた調和級数 $\sum_{r=1}^{\infty} 1/r^{\alpha}$ は $\alpha = 1$ のとき無限大に発散しますから, 式(3.11)の確率の和は 1 にならなくなってしまいます. よって級数が収束するためには, 少なくともある順位からは $\alpha > 1$ になっているはずです. 実際に図 3.6(a) でもみられるように, 一般に $r \to 1$ と $r \to \infty$ では巾乗則の傾き α が異なっており, この現象は double power-law [71]とよばれていま

---

*21 George Kingsley Zipf (1902–1950) は, 1935 年に[66]で Zipf の法則を示しました. ただし, この法則は Zipf が最初に発見したわけではなく, 1916 年にはフランスの速記者 Estoup によって発見されていました[67, 54]. 最近になり, Estoup の孫にあたる方によって, この事情についての論文が出版されています[68].

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

す. この説明として, 語彙を核となる語彙と周辺の語彙に分ける生成モデル[72]などが研究されています [*23].

*23　数学的には, 最近オックスフォード大学の Caron らによって, Lévy 測度 $\rho(w) = 1/\Gamma(1-\sigma)w^{-1-\tau}\gamma(\tau-\sigma, cw)$ ($\gamma()$ は第一種不完全ガンマ関数) をもつ一般化 BFRY (Bertoin-Fujita-Roynette and Yor) 過程を考えると, これを和が 1 になるように正規化した正規化 GBFRY 過程がこの double power-law をもち, 言語に非常によく当てはまることが示されています[73].

コラム：単語の前処理について

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

そこで, 一般にたとえば数字を`#`で置き換えたり, 特殊文字を削除したり, `aaaaaaa` のような同じ文字の連続を 3 文字に縮約して `aaa` とするなどの前処理を行います. こうすると, たとえば `No.123` は `no.###`に, `2021.3.28` は `####.#.##` に, `Goooooooal` は `goooal` になります. また, 大文字と小文字の違いで別の単語と認識されないよう, ここに示したようにすべて小文字に直すこともよく行われる処理の一つです.[*24]

前処理には決まった手続きがあるわけではありませんが, 英語の場合は教科書 FSNLP [15]のサイトにある `sed.strip`[*25] などを使うのが簡単です. 日本語の場合は, 後で紹介する mecab-NEologd のサイト [*26] に正規化処理についてまとめられているほか, この処理を pip でインストールできるパッケージにした `neologdn`[*27] が公開されています. 表 3.2 は, こうした前処理を行った上で単語を数えたものです.

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

---

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

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

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

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

3.3　単語の統計的フレーズ化

3.1 節で述べたように, 英語では “New York”, “with respect to”, 日本語では「基本 的」,「富嶽 三 十 六 景」のように, 空白で区切られていても, 実際は一つの単語として働く表現は多く存在します. また,「御 樋 代 木 奉 曳式」[*28] のように特殊な言葉のために形態素解析が失敗して多くの単語に分割されてしまっている例もありますし,「高 エネルギー リン 酸 化合 物」「お ジャ 魔女 どれ み ♯」[*29] のような長い単語列を固有名詞として認識したい場合もあるでしょう [*30]. このような場合には, どうすればいいのでしょうか.

日本語の場合, 最も簡単な方法は, 多くの新語や固有名詞に対応している MeCab 用の辞書 mecab-ipadic-NEologd [75] [*31] を用いることです. これには多くの固有表現が含まれており, 一般的な語については一語として認識して形態素解析されます. より一般には, 認識したい上記のような固有表現の正解データを多数, 人手で用意しておけば, そこから教師あり学習によって, 新しいテキストに対して該当個所を固有表現として認識してくれる識別器を学習する**固有表現認識** (Named Entity Recognition, NER) とよばれる方法でフレーズを認識することができます (本書では正解データが必要な教師あり学習は基本的に扱いませんので, 興味のある方は[21, Sec.5.5]などを参照してください).

ただし, 前者の辞書は人手で更新されていたものですから, 有名な固有名詞はカバーされているものの,「t 細胞受容体」「ディビダーク式カンチレバー工法」のような専門的な名前にすべて対応することは, 原理的に不可能です. また, 後者の NER はあくまで正解データとして与えた表現およびそれに近い表現を認識するものですから, あらゆるフレーズに対応するには, 厖大な量の「正解デー

---

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

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

*30　日本語学では, “中小企業経営者像” のようにその場で名詞として働く句は「臨時一語」[74] とよばれています (青山学院大学名誉教授 近藤泰弘先生のご指摘による). ただし, これは統語的に形成されるもので, いま考えている, 繰り返し使われて実質的に「単語」として働く表現とは異なります.

*31　`https://github.com/neologd/mecab-ipadic-neologd` ただし, 現在は更新は停止しています.

タ」を作って更新する必要があり, その際の基準も一意とは限らないために現実的ではありません.

しかし, よく考えると, “San” の後には必ず “Francisco” が続くのであれば, “San Francisco” を一つの単語として認識してもよいはずです. 同様に, 「宮内」の後には「庁」が続く確率が非常に高く, また「楽部」の前は「宮内 庁」であることが非常に多いのであれば, 「宮内庁」「宮内庁楽部」をそれぞれ単語とみなしてもよいでしょう.

このような考え方に基づき, 3.5.2 節で説明する単語のベクトル化の方法である有名な Word2Vec を発明した Mikolov ら [76]は, 空白で区切られた単語をそのままベクトル化するのではなく, 文脈に応じて “and new_york city council ..” のように ‘_’ によって単語を繋げて自動的にフレーズ化してから Word2Vec を計算する方法を示しました. *32

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

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

ここで, $n(v,w)$ および $n(v), n(w)$ はそれぞれ単語バイグラムおよびユニグラムの頻度で, δ は頻度 $n(v,w)$ の小さいバイグラムのスコアを下げるための割り引き係数です. たとえば $v=$ `san`, $w=$ `francisco` で, $n(v,w)=100$, $n(v)=100$, $n(w)=120$ だったとき *33, $\delta=10$ であれば

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

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

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

*32 この方法は, すでに 1992 年には IBM の Brown ら[77]によって提案されています.

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

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

ただし, Mikolov らの論文 [76]で提案されている式(3.12)のスコアには注意が必要です [*34]. テキスト全体の長さを N とすると, バイグラム (v, w) およびユニグラム v, w の確率はそれぞれ

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

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

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

となり, このスコアは**学習テキストの長さ N に依存します**. さらに, **割り引き係数 δ の大きさも N によって変えなければならない**, ということがわかります. テキストを固定して閾値を探索するのであれば問題ありませんが [*35], スコアに絶対的な意味がないために探索が難しく, 違うテキストを同じ基準で前処理することもできなくなってしまいます. この欠点はスコアを N 倍すれば解消しますが, その場合も δ/N の部分は残るため, 適切な δ の設定は困難になります.

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

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

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

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

---

*34　以下の内容は, 本書のオリジナルです.

*35　実際には, フレーズ化を数パス繰り返す中で単語が結合され, N の値も変わってきてしまいますので, もとの方法では適切な閾値の設定はさらに困難になります.

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

図 3.7: 正規化自己相互情報量 (NPMI) に基づいて 2 単語を統計的にフレーズ化した例. 認識されたフレーズを [] で示しました. 学習には日本語版 text8 コーパスを使用しています.

と, 本質的に同じ意味を持っていることがわかります. 式(3.16)の PMI は, v と w が共起する確率 $p(v,w)$ が, v と w が独立だった場合の確率 $p(v)p(w)$ と比べて何倍なのか (の対数) を表しています. これは統計的に v と w の相関を見るためには理想的な量で, 情報理論に基づいて 1989 年にベル研究所の Church ら [78]によって提案されました.

ただし, PMI には

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

という欠点があります. そこで, 5 章でも説明するように, PMI をその最大値である, $-\log p(v,w)$ で割って正規化した**正規化自己相互情報量** (Normalized PMI, **NPMI**) [79]

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

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

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

の値をとり, v と w が完全に相関しているとき 1, 完全に逆相関しているとき -1

の値をとるという, 大変よい性質を持っています. 式(3.17)は, 頻度 $n()$ を使えば

$$\begin{aligned}(3.19)\quad \mathrm{NPMI}\,(v,w) &= \log\frac{p(v,w)}{p(v)p(w)}\Big/\Big(-\log p(v,w)\Big)\\ &= \log\left(\frac{n(v,w)}{\cancel{N}}\cdot\frac{\cancel{N}}{n(v)}\cdot\frac{N}{n(w)}\right)\Big/\left(-\log\frac{n(v,w)}{N}\right)\\ &= \frac{\log N+\log n(v,w)-\log n(v)-\log n(w)}{\log N-\log n(v,w)}\end{aligned}$$

と表すことができます. このとき, NPMI の仕組みによって, 式(3.12)で $n(v,w)$ が小さいバイグラムのスコアを下げるための**ヒューリスティックな割り引き係数 δ は不要になっている**ことに注意してください. この計算は, サポートサイトにある `phraser.py` を使って,

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

のように実行すると簡単に行うことができます.

この NPMI が閾値, たとえば 0.5 以上になった 2 単語をフレーズとみなして日本語 text8 に適用した例を図 3.7 に示しました. ほとんどの語はそのままですが, NER を使わなくても教師なしで, 「ニューディール_政策」「上杉_謙信」「さ_れ」などがフレーズとして自動的に認識されていることがわかります.

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

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

しかし一方で, たとえば「逆転写酵素」は「酵素」の一種を意味しているという情報は失われてしまうため, こうしたフレーズ化は, フレーズの頻度が充分高

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

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

い場合にのみ行った方がよいでしょう. 上記の phraser.py では, デフォルトではバイグラム頻度が 10 以上の場合にのみフレーズ化を行う設定になっており, 閾値も含めてオプションで変えることができます. 使い方は,

```
% phraser.py
```

をそのまま実行するか, スクリプトの中身を読んでみてください.

3.4　単語nグラム言語モデル

単語についても, 2.5 節の文字の場合と同じように単語nグラム言語モデルを考えることができます. 現在は文の確率を計算するには, Transformer や LSTM などの深層学習モデルを利用するのが一般的ですが[*36], 本書で扱うさまざまな統計モデルの基礎になりますので, 単純で理由のわかっている, この最も基本的なモデルについて考えていくことにしましょう.

単語が直前の単語だけに依存するバイグラム言語モデルを考えるとすると, 文s=「銀河 の お祭 です」の確率は式(2.46)と同様に,

$$p(s) = p(\text{銀河}|\hat{}\,)\,p(\text{の}|\text{銀河})\,p(\text{お祭}|\text{の})\,p(\text{です}|\text{お祭})\,p(\$|\text{です}) \tag{3.20}$$

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

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

```
from collections import defaultdict
def parse (file):
    EOS = "_EOS_"
    freq = {}
    with open (file, 'r') as fh:
        for line in fh:
            words = line.rstrip('\n').split() # 空白で分割
            words.insert (0, EOS); words.append (EOS)
            T = len(words)          # ↑文頭/文末文字を追加
            for t in range(T-1):
                w = words[t]
```

*36　ただし, こうした深層学習モデルがどうしてうまく単語を予測できるのかは, まだ充分わかっていないのが現状です.

```
                    v = words[t+1]
                    if not (w in freq): # freq[w] がなければ作成
                        freq[w] = defaultdict(int)
                    freq[w][v] += 1     # 頻度に 1 を加える
        return freq
```

のように関数を定義してから,

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

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

```
freq["銀河"]["ステーション"]
⇒ 5
```

のようになります.

式(3.20)のそれぞれのバイグラムの条件つき確率は, 頻度を表す関数 $n()$ を使って, 最も簡単には式(2.4)のような最尤推定値

$$\widehat{p}(v\,|\,w) = \frac{n(w,v)}{\sum_v n(w,v)} = \frac{n(w,v)}{n(w)} \tag{3.21}$$

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

ただし, 式(3.21)を使った単語バイグラム確率の計算には, すぐに問題があることがわかります. ここまでにみたように, Zipf の法則から頻度のほとんどは一

部の単語に集中しており, 大部分の単語は低頻度なのでした. すると, たとえば「銀河」の次に「旅行」が現れる確率は, 式(3.21)に従えば

$$p(\text{旅行}\,|\,\text{銀河}) = \frac{n(\text{銀河},\text{旅行})}{n(\text{銀河})} \tag{3.22}$$

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

$$p(\text{旅行}\,|\,\text{銀河}) = \frac{n(\text{銀河},\text{旅行})}{n(\text{銀河})} = \frac{0}{n(\text{銀河})} = 0 \tag{3.23}$$

となり, 「銀河旅行」の確率は式(2.20)の確率の連鎖則から,

$$p(\text{銀河　旅行}) = \underbrace{p(\text{旅行}\,|\,\text{銀河})}_{=0} \cdot p(\text{銀河}) = 0 \tag{3.24}$$

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

文字の場合は種類が少ないため, 文字バイグラムではゼロ頻度問題はあまり深刻になりませんでしたが[*37], 単語は種類が多いため, バイグラムでも, ほとんどの確率が 0 になってしまうという問題が生じます.

ゼロ頻度問題を避けるために, 2 章の文字nグラムモデルでは式(2.52)の**加算平滑化**という手法を用いました. 語彙の総数を V とおくと, これは式(3.21)の代わりに, すべての頻度に小さな値 α を足して

$$p(v\,|\,w) = \frac{n(w,v)+\alpha}{\sum_{v=1}^{V}\big(n(w,v)+\alpha\big)} = \frac{n(w,v)+\alpha}{n(w)+V\alpha} \tag{3.25}$$

とするものです. こうすると, $n(w,v)=0$ の場合でも, 確率は

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

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

---

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

ところが, 単語の場合は式(3.25)を使ってもまだ問題があることがわかります. たとえば, 単語 w と v は「銀河 鉄道」のように常に wv の形で出現しており, $n(w,v)=10$, $n(v)=n(w)=10$ だったとしましょう. $\alpha=0.01$, $V=10000$ とすると, 式(3.25)は

$$p(v|w)=\frac{n(w,v)+\alpha}{n(w)+V\alpha}=\frac{10+0.01}{10+10000\cdot 0.01}=\frac{10.01}{10+100}=0.091 \tag{3.26}$$

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

これはもちろん, 「現れなかった単語も含め, **すべての単語の頻度に同じ α を足す**」ということが原因です. 「集団」のような頻度の高い語も, 「セロリ」のような頻度の低い語も同じ α が足されるため, たまたま「銀河」の後に続いたことがなければ「銀河集団」と「銀河セロリ」が同じ確率を持つことになり, それらの確率の総和である式(3.25)の $V\alpha$ の部分が非常に大きくなってしまうからです.

言うまでもなく, これは誤りです. 文脈となる語 w と予測したい語 v を分けて考えると, v に対応する α の値は, 大まかには v の確率 $p(v)$ に比例する値 α_v とするのがよいでしょう. ただしよく考えると, 単に $p(v)$ に比例するのではなく, たとえ同じ頻度でも, 多くの語の後に続きやすい「前」のような語の α_v は大きく, 他の語の後に続きにくい「あの」のような語の α_v は小さくするべきだと考えられます.[*38] そして, 決めるべきパラメータは $\alpha_1,\alpha_2,\cdots,\alpha_V$ で, この場合 10,000 個もあります. この $\alpha_1,\alpha_2,\cdots,\alpha_V$ はどうやって決めればよいのでしょうか.

こういった複雑な問題を解くには, 今までのような発見的な方法では限界があります. そこで, より原理的に考えてみることにしましょう. 以下の内容は, 5 章で文書のモデルを考える際の基礎にもなっています.

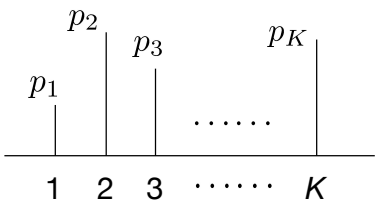


図 3.8: K 次元の多項分布 $\mathbf{p}$. 確率の和は $\sum_{k=1}^{K} p_k = 1$ になっています.

3.4.1　ディリクレ分布

これまで, 式(3.25)のような確率の由来については考えず, 単に「頻度やそれに値を足したものを和が 1 になるように正規化する」ことしか行ってきませんでした. 式(3.25)のような確率は, すべての単語 $v = 1, 2, \cdots, V$ について考えると総和が 1 になる**確率分布**ですから, そもそも確率分布を生み出すことのできる**確率モデル**について考えてみましょう.

いま, 図 3.8 のような K 次元の確率分布を

$$\mathbf{p} = (p_1, p_2, \cdots, p_K) \qquad \left(p_1, p_2, \cdots, p_K \geq 0, \sum_{k=1}^{K} p_k = 1 \right) \tag{3.27}$$

とします. K は次元の数で, 言語の場合には K は実際には 10,000 や 100,000 といった大きな値になります. 確率 $p_1, p_2, \cdots, p_K$ を直接指定するこの確率分布は, もっとも基本的な分布で, 正確には**多項分布**または離散分布とよびます.[*39]

たとえば $K = 3$ のとき,

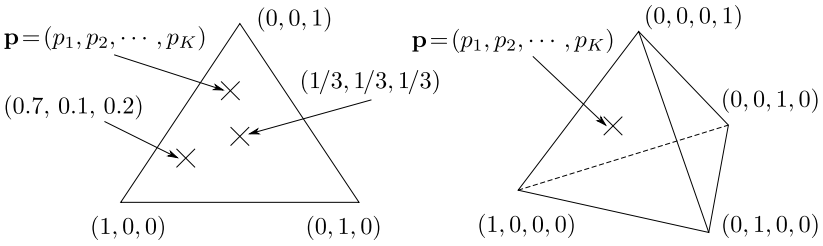
$$\mathbf{p} = (0.7, 0.1, 0.2)$$

は $\mathbf{p}$ の 1 つの例です. より一般に $\mathbf{p} = (p_1, p_2, p_3)$ は, これをベクトルとみなせば, 図 3.9(a) のように正三角形の内部にあると考えることができます. こうした図形を, **単体** (simplex) といいます. $\mathbf{p} = (1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ の各分布

---

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

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



(a) $K=3$ の場合　　(b) $K=4$ の場合

図 3.9: 単体とその上の多項分布 $\mathbf{p}$. 単体の各端点は, そのカテゴリだけが確率 1 で出る確率分布になっています.

はそれぞれ, この単体の 3 つの角に対応し, $\mathbf{p}=\left(\frac{1}{3},\frac{1}{3},\frac{1}{3}\right)$ は単体の中心に対応しています. なお $K=4$ の場合は, 単体は図 3.9(b) のような正四面体になります.

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

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

のとき, カテゴリ 1,2,3,4 を確率分布 $\mathbf{p}$ に従ってランダムに 5 回選んだ結果が

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

だったとすれば [*40], Y の確率は

$$p(Y\,|\,\mathbf{p}) = p_4\cdot p_2\cdot p_1\cdot p_1\cdot p_2 = p_1^2\cdot p_2^2\cdot p_3^0\cdot p_4^1 = \prod_{k=1}^{4} p_k^{n_k} \tag{3.28}$$

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

*40 ここでは, Y にテキストのように順番を考えています. 順番を考えない場合は, 可能な組み合わせの総数である多項係数 $\binom{5}{2\ 2\ 0\ 1}=\frac{5!}{2!\,2!\,0!\,1!}$ がかかることになりますが, この係数はパラメータ $\mathbf{p}$ を含んでいませんので, $\mathbf{p}$ に関する推定値は同じになります.

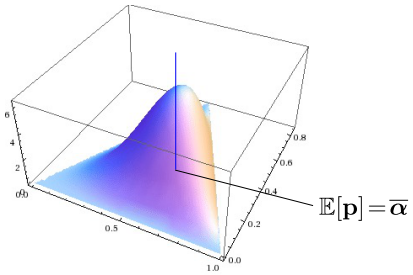


図 3.10: 単体上のディリクレ分布 $\mathrm{Dir}(\mathbf{p}|\boldsymbol{\alpha})$ とその期待値 $\mathbb{E}[\mathbf{p}]=\overline{\boldsymbol{\alpha}}$.

こうした $\mathbf{p}$ を生成する単体上の最も簡単な確率分布として, 次の式で表される**ディリクレ (Dirichlet) 分布**があります. *41

$$\mathrm{Dir}(\mathbf{p}|\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} p_k^{\alpha_k-1} \qquad (\textbf{ディリクレ分布 (簡易版)}) \tag{3.29}$$

図 3.10 に, この分布の形を示しました. $\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_K)$ は分布のパラメータで, $\alpha_k \geq 0$ $(k=1,2,\cdots,K)$ は非負の実数です *42. $\mathbf{p}$ はそれ自身確率分布ですから, ディリクレ分布は「確率分布を生成する確率分布」ということになります. $\mathbf{p}$ についての関数であることを示すため, 式(3.29)では $\mathrm{Dir}(\mathbf{p}|\boldsymbol{\alpha})$ と書きましたが, 以下では簡単のため, $\mathrm{Dir}(\boldsymbol{\alpha})$ と略記することにします. ディリクレ分布から生成される $\mathbf{p}$ の期待値 $\mathbb{E}[\mathbf{p}]$ は, $\boldsymbol{\alpha}$ を和が 1 になるように正規化した

$$\mathbb{E}[\mathbf{p}] = \overline{\boldsymbol{\alpha}} = \left(\frac{\alpha_1}{\sum_k \alpha_k}, \cdots, \frac{\alpha_K}{\sum_k \alpha_k}\right) \qquad (\textbf{ディリクレ分布の期待値}) \tag{3.30}$$

となります. ここで, $\sum_{k=1}^{K}=\sum_k$ と略記しました. 証明については, 付録 A.2 を参照してください.

ディリクレ分布からのサンプルは, Python では `numpy.random.dirichlet()` で作ることができます. または, ガンマ分布 $\mathrm{Ga}(\alpha_k,1)$ からのサンプル *43

---

*41　この名前は, 正規化定数となる式(3.38)の積分を示した, 旧フランス領で生まれたドイツの数学者 Peter Gustav Lejeune Dirichlet (1805–1859) にちなむものです [80].

*42　$\alpha_k=0$ のときは, 対応する p_k はつねに 0 であると約束します.

*43　この 1 はスケールを表しているだけですので, 同一であれば別の値でも問題ありません. ガ

$$\gamma_k \sim \mathrm{Ga}(\alpha_k, 1) \quad (k = 1, 2, \cdots, K) \tag{3.31}$$

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

$$\mathbf{p} = \frac{1}{\sum_k \gamma_k}(\gamma_1, \gamma_2, \cdots, \gamma_K) \sim \mathrm{Dir}(\alpha_1, \alpha_2, \cdots, \alpha_K) \tag{3.32}$$

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

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

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

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

から生成されたランダムな多項分布 $\mathbf{p}$ をプロットすることができますので, 試してみましょう. 図 3.11 に, いくつかの $\boldsymbol{\alpha}$ の値について, こうして $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成した多項分布 $\mathbf{p}$ の例を示しました. $\alpha_k = 1$ の場合は $\mathbf{p}$ は自由な形になりますが, $\alpha_k > 1$ の場合は $\mathbf{p}$ は期待値である $\overline{\boldsymbol{\alpha}}$ に近く, $\alpha_k < 1$ の場合は $\mathbf{p}$ は少数の p_k の値だけが大きく, 他がほとんど 0 に近い疎な分布となることがわかります. 言語の場合は, ほとんどはこの場合に対応しています.

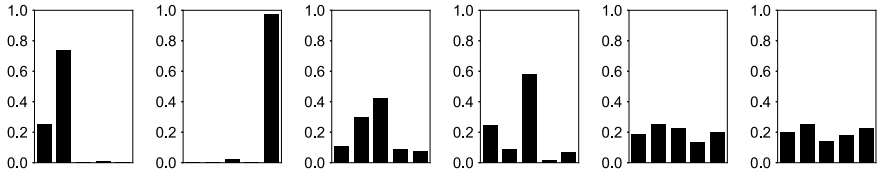
ガンマ関数とディリクレ分布

式(3.29)では比例 $\propto$ を使ってディリクレ分布を示しましたが, 正確に書くと, ディリクレ分布の定義は

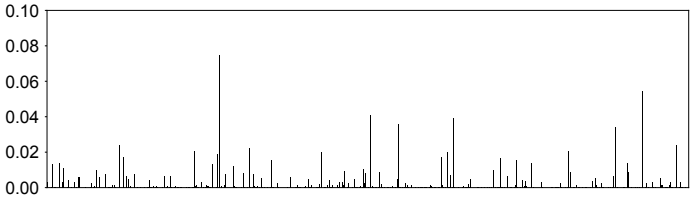
$$\mathrm{Dir}(\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} \qquad (\textbf{ディリクレ分布 (正式版)}) \tag{3.33}$$

となります. この定義式(3.33)は一見いかつい形をしていますが, ガンマ関数 $\Gamma(x)$ を含む前半部分は, 可能な $\mathbf{p}$ 全体についての積分を 1 にするための正規化定数で, 本質的には, ディリクレ分布は式(3.29)で表される**簡単な分布**であることに注意してください. $\Gamma(x)$ $(x \in \mathbb{R})$ は階乗関数 $x! = x(x-1)(x-2)\cdots 1$ の連

ンマ分布からのサンプルは, Python では `numpy.random.gamma()` で作ることができます. 自分で $[0, 1)$ の乱数からガンマ分布に従う乱数を計算する方法は複雑ですので, 必要な方は乱数生成の専門書[81]を参照してください. サポートページに, 筆者が使っている C 言語による実装 `gamma.{c,h}` が置いてあります.



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



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

図 3.11: ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ からランダムにサンプルした多項分布 $\mathbf{p}$ の例.

続値への一般化とみることができる関数で,

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \qquad (\textbf{ガンマ関数}) \tag{3.34}$$

で定義されます. 式(3.34)を部分積分することで,

$$\Gamma(x+1) = x\Gamma(x) \tag{3.35}$$

を示すことができますので, 確かめてみましょう (→演習 3–6). 式(3.35)から, x が整数であれば

$$\begin{aligned} \Gamma(x) &= (x-1)\Gamma(x-1) = (x-1)(x-2)\Gamma(x-2) \\ &= (x-1)(x-2)\cdots\cdot 2\cdot 1 \\ &= (x-1)! \end{aligned} \tag{3.36}$$

が成り立ちます. x が整数でない場合は, x 以下の最大の整数を $n = \lfloor x \rfloor$ とおき, $x = n + \alpha$ と書けば

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

$$= (n+\alpha-1)(n+\alpha-2)\cdots\cdot(\alpha+1)\cdot\alpha$$
$$= (x-1)(x-2)\cdots\cdot(\alpha+1)\cdot\alpha$$

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

式(3.33)の正規化定数は, この $\Gamma(x)$ を使って

$$\int \prod_{k=1}^{K} p_k^{\alpha_k-1} d\mathbf{p} = \int_0^1\int_0^1\cdots\int_0^1 p_1^{\alpha_1-1}p_2^{\alpha_2-1}\cdots p_K^{\alpha_K-1} dp_1 dp_2\cdots dp_{K-1} \tag{3.38}$$
$$= \frac{\prod_k \Gamma(\alpha_k)}{\Gamma(\sum_k \alpha_k)} \qquad (\textbf{ディリクレ分布の積分公式})$$

の積分から得られるものです [*44]. この積分は, 直感的には, 図 3.10 のような単体上の曲面の下の体積を求めることに相当しています. 式(3.38)の積分は大学教養範囲の数学 (解析) を必要としますので, 詳しくは付録 A.1 を参照してください. 上のように, 以下本書では, 単体上の積分 $\int_0^1\int_0^1\cdots\int_0^1 dp_1 dp_2\cdots dp_{K-1}$ を略して $\int d\mathbf{p}$ と書くことにします.

ディリクレ分布は, この単体上の確率分布です. ディリクレ分布はパラメータ $\boldsymbol{\alpha}$ の値によって, さまざまな形をとります. 図 3.12 に, $\boldsymbol{\alpha}$ の値とそれに対応するディリクレ分布の形を示しました [*45]. 式(3.29)からわかるように, $\alpha=(1,1,\cdots,1)$ のとき

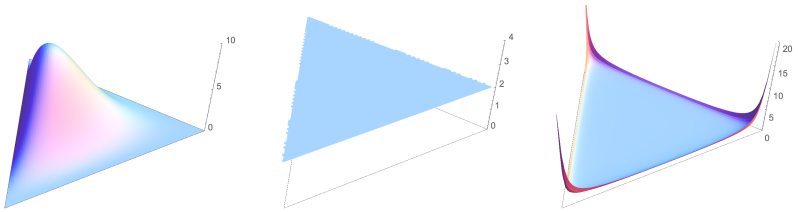
$$\mathrm{Dir}(\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} p_k^{1-1} = \prod_{k=1}^{K} p_k^0 = 1 \tag{3.39}$$

ですから, 確率分布は図 3.12(b) のように単体上の一様分布になります. $\alpha_k >$

---

*44 $p_1, p_2, \cdots, p_K$ は独立ではなく, $\sum_{k=1}^{K} p_k = 1$ すなわち $p_K = 1-p_1-\cdots-p_{K-1}$ となるため, 積分は $K-1$ 個の自由度に関するものになります. 積分範囲も単体上に限られるため, 本書では正確には $\int_0^1\cdots\int_0^1$ は, $\int_0^1\cdots\int_0^1 \mathbb{I}\left(\sum_{k=1}^{K} p_k = 1\right)$ の略記だと考えてください.

*45 標準的な三次元プロットでは境界がガタガタになるため, これは *Mathematica* で正三角形の内部を明治屋のロゴのように 4 つに再帰的に分割して, 自前でポリゴンを描画したものです. サポートサイトの `Dirichlet.nb` に, 使用した *Mathematica* のコードがあります.



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

図 3.12: さまざまな $\boldsymbol{\alpha}$ によるディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ の確率密度関数.

1 のときは図 3.12(a) のように上に凸な, $\alpha_k < 1$ のときは図 3.12(c) のように下に凸な分布になっており, それぞれ期待値に近い, あるいはどれかの p_k だけが大きい疎な $\mathbf{p}$ を生み出すことになります.

サイコロ工場とディリクレ分布　図 3.8 のような多項分布 $\mathbf{p}=(p_1,p_2,\cdots,p_K)$ は, 「ゆがんだ K 面サイコロ」のようなものだと考えることもできます. サイコロの各面 k の大きさが確率 p_k に対応しており, このサイコロを振ると, p_k に比例した確率で値 k が出るというわけです.

このとき式(3.29)のディリクレ分布は図 3.13 に示したように, 様々な K 面体サイコロ $\mathbf{p}$ を生産する「サイコロ工場」のようなものだと考えてもいいでしょう [82]. この工場からは様々なゆがみを持った K 面体サイコロ $\mathbf{p}$ が生産されますが, 工場には特有の傾向があり, $\mathbf{p}$ がほとんど一様なサイコロを生産する工場や, $\mathbf{p}$ の値が大きく異なるサイコロを生産する工場もあります. ある工場から生産されたサイコロ $\mathbf{p}$ を多く集めて計測すれば, この工場の持つ傾向 $\boldsymbol{\alpha}$ を推測することができます.

3.4.2　ディリクレ分布と多項分布

$\mathbf{p}$ の分布として, 式(3.29)を最も簡単な分布だとしたのには理由があります. いま $K=3$ で, $\mathbf{p}=(p_1,p_2,p_3)$ の分布が $\mathrm{Dir}(\boldsymbol{\alpha})$ に従っているとしましょう.

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

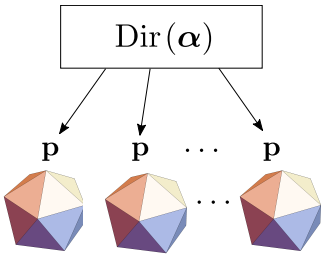


図 3.13: 「サイコロ工場」としてのディリクレ分布. $\boldsymbol{\alpha}$ の値によって, それぞれ異なる歪みを持った多面体サイコロ $\mathbf{p}$ が生産されます.

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

$$p(\mathbf{p}|Y) \propto p(Y|\mathbf{p})\,p(\mathbf{p}) \tag{3.40}$$

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

$$p(\mathbf{p}) \propto \prod_{k=1}^{3} p_k^{\alpha_k - 1} \tag{3.41}$$

です. $\mathbf{p}$ から Y が得られる確率 $p(Y|\mathbf{p})$ は, 式(3.28)と同様にして

$$p(Y|\mathbf{p}) = p_2 \cdot p_3 \cdot p_1 \cdot p_2 \cdot p_2 = p_1^1 \cdot p_2^3 \cdot p_3^1 = \prod_{k=1}^{3} p_k^{n_k} \tag{3.42}$$

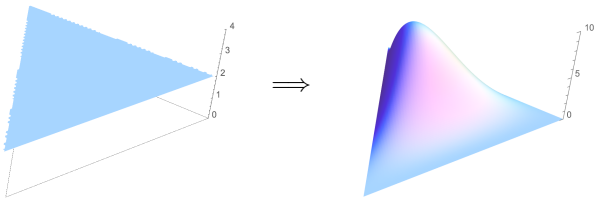
となります. ここで n_k は Y の中で k が出た回数で, $n_1=1, n_2=3, n_3=1$ です. これをまとめて, $\mathbf{n}=(n_1,n_2,n_3)=(1,3,1)$ と書くことにしましょう.

式(3.41)と式(3.42)をあわせると, Y が与えられたときの $\mathbf{p}$ の分布は,

$$p(\mathbf{p}|Y) \propto p(Y|\mathbf{p})\,p(\mathbf{p}) = \prod_{k=1}^{3} p_k^{n_k} \times \prod_{k=1}^{3} p_k^{\alpha_k - 1} = \prod_{k=1}^{3} p_k^{\alpha_k + n_k - 1} \tag{3.43}$$

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

$$\begin{aligned} &\mathrm{Dir}(\alpha_1+n_1, \alpha_2+n_2, \alpha_3+n_3) \\ &= \mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n}) \quad (\mathbf{n}=(n_1,n_2,n_3)) \qquad (\textbf{ディリクレ事後分布}) \end{aligned} \tag{3.44}$$



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

図 3.14: ディリクレ事前分布と事後分布の例.

であることがわかります.

つまり, $\mathbf{p}$ の事前分布を式(3.29)のディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ にすると, Y を観測した後の事後分布はパラメータを $\boldsymbol{\alpha}+\mathbf{n}$ に更新した, 同じディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n})$ になるわけです.[*46] この様子を, 図 3.14 に示しました. $\mathbf{p}$ の事前分布が図 3.14(a) のように一様分布, すなわち $\mathrm{Dir}(1,1,1)$ だったとすると, 式(3.43)から Y を観測した後の $\mathbf{p}$ の事後分布は $\mathrm{Dir}(1+1,1+3,1+1)=\mathrm{Dir}(2,4,2)$ となり, 図 3.14(b) に示したような分布になります.

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

$$\overline{\boldsymbol{\alpha}} = \left(\frac{\alpha_1}{\sum_k \alpha_k}, \frac{\alpha_2}{\sum_k \alpha_k}, \cdots, \frac{\alpha_K}{\sum_k \alpha_k} \right) \tag{3.45}$$

でしたから, この事後分布の期待値は

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

となります. すなわち Y を観測した下では, 次のカテゴリ $1,2,3$ が出る確率はそれぞれ $0.25, 0.5, 0.25$ になります.

よって一般に, K 次元の多項分布 $\mathbf{p}$ からの観測値 Y としてそれぞれの値 k $(k=1,2,\cdots,K)$ が n_k 回観測されたとき, $\mathbf{p}$ の事前分布を $\mathbf{p} \sim \mathrm{Dir}(\boldsymbol{\alpha})$ とすれ

---

*46　こうした性質を, 確率分布の**共役**(きょうやく)性といいます. 多項分布とディリクレ分布は, 共役な分布です. 多項分布に共役な分布は他にもありますが, たとえばニューラルネットでよく使われている Softmax 関数 (135 ページ) は多項分布と共役ではなく, このように簡単に事後分布を計算することはできません. なお本来の漢字は「共軛」で, 軛(くびき) とは, 馬車などで動物の首を結びつけて一緒に動かすための棒のことです.

ば, 頻度ベクトルを $\mathbf{n}=(n_1,\cdots,n_K)$ として, 事後分布は

$$\mathbf{p}|Y \sim \mathrm{Dir}(\boldsymbol{\alpha}+\mathbf{n}) \tag{3.46}$$

になり, その期待値は

$$\mathbb{E}[p_k|Y] = \frac{\alpha_k+n_k}{\sum_k(\alpha_k+n_k)} = \frac{n_k+\alpha_k}{N+\alpha} \qquad (\textbf{ディリクレ平滑化}) \tag{3.47}$$

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

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

式(3.47)のディリクレ平滑化と式(3.25)の加算平滑化を比べると, 加算平滑化は, $\mathbf{p}$ に均一なディリクレ分布

$$\mathbf{p} \sim \mathrm{Dir}(\alpha,\alpha,\cdots,\alpha) \tag{3.48}$$

すなわち, $\alpha_k=\alpha$ とした場合のディリクレ平滑化と等しいことがわかります[*47]. こうして, 今まで発見的に定義した加算平滑化を, ディリクレ平滑化の特別な場合として導くことができました.

ハイパーパラメータ $\boldsymbol{\alpha}$ の推定

さて, 上のように $\mathbf{p}$ をディリクレ分布でモデル化することの意義は何でしょうか. それは, これによって**データから最適な $\boldsymbol{\alpha}$ を推定できる**ということです.

われわれは, $\mathbf{p}$ 自体を直接観測することはできません. しかし, $\mathbf{p}$ からサンプルされた各カテゴリの頻度である $\mathbf{n}$ が毎回ほぼ均一な値であれば, $\boldsymbol{\alpha}$ は図 3.11(c) のようにほぼ同じで, 1 より大きい値だと推測できるでしょう. いっぽう, $\mathbf{n}$ の値がどれかの次元に偏っていれば, $\boldsymbol{\alpha}$ は図 3.11(a) のように 1 より小さいと考えられます. また, どれかの n_k の値がいつも大きくなっていれば, 対応する p_k の値, したがって α_k も大きいと考えられます. 3.4.1 節のサイコロ工場のメタ

*47 ここで $\alpha=1$ とした最も簡単な場合を, **ラプラス平滑化**とよびます. ラプラス平滑化は, 確率論の初期に賭けの計算のために, ラプラスによって導入されました[24].

ファーを使えば, 生産されたサイコロ $\mathbf{p}$ の各面の面積を正確に測定することができなくても, $\mathbf{p}$ をランダムに転がした結果の集計 $\mathbf{n}$ さえあれば, 工場のパラメータ $\boldsymbol{\alpha}$ を推測できるはずだ, というわけです.

そこで, こうした $\mathbf{n}$ が複数観測されたとき [*48], そこから共通する $\boldsymbol{\alpha}$ を推定する問題を考えてみましょう. すなわち, D 個の頻度ベクトル

$$\mathcal{D} = \{\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_D\} \tag{3.49}$$

から, $\boldsymbol{\alpha}$ を求めることを考えてみます.

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

ステップ 1. 多項分布 $\mathbf{p} \sim \mathrm{Dir}(\boldsymbol{\alpha})$ をサンプル. $\mathbf{n}=(0, 0, \cdots, 0)$.

ステップ 2. For $i = 1, 2, \cdots, N$,

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

こうした, データが生成された過程のモデルを**生成モデル**といいます.[*49]

この生成モデルに従えば, 多項分布 $\mathbf{p}$ とそこからの観測値 $\mathbf{n}$ が生成される確率は, 次のように表すことができます.

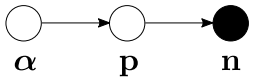
$$p(\mathbf{n}, \mathbf{p}|\boldsymbol{\alpha}) = p(\mathbf{n}|\mathbf{p})\, p(\mathbf{p}|\boldsymbol{\alpha}) \tag{3.50}$$

第 1 項は上のステップ 2 に, 第 2 項は上のステップ 1 に対応しています. この関係をわかりやすくするために, 模式的に図 3.15(a) のように表してみましょう. こうした図を**グラフィカルモデル**といい, ○で表される変数間の依存関係が $\rightarrow$ で表されています. ○は未知の変数を, ●は観測された変数すなわちデータを表します. ここでは $\boldsymbol{\alpha}$ から $\mathbf{p}$ が生成され, $\mathbf{p}$ から $\mathbf{n}$ が生成されるため, $\boldsymbol{\alpha} \rightarrow \mathbf{p} \rightarrow \mathbf{n}$ という生成モデルとなり, これが式(3.50)を表しています.

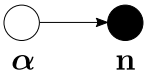
---

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

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



(a) ディリクレ多項分布.



(b) ポリア分布. $\mathbf{p}$ が期待値をとって積分消去されています.

図 3.15: ディリクレ多項分布とポリア分布のグラフィカルモデル.

さて, 式(3.50)の第 1 項は, カテゴリ k が出る確率が p_k なのですから

$$p(\mathbf{n}|\mathbf{p}) = \prod_{k=1}^{K} p_k^{n_k} \tag{3.51}$$

です. また第 2 項は, $\mathbf{p}$ はディリクレ分布に従うので, 式(3.33)より

$$p(\mathbf{p}|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k - 1} \tag{3.52}$$

です. よって, 式(3.50)は

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

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

$$\begin{aligned} p(\mathbf{n}|\boldsymbol{\alpha}) &= \int p(\mathbf{n}, \mathbf{p}|\boldsymbol{\alpha}) d\mathbf{p} \\ &= \int \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p} = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \underbrace{\int \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p}}_{(\star)} \end{aligned} \tag{3.53}$$

となります. ここで, $(\star)$ は式(3.38)のディリクレ分布の積分公式から,

$$\int \prod_{k=1}^{K} p_k^{\alpha_k + n_k - 1} d\mathbf{p} = \frac{\prod_k \Gamma(\alpha_k + n_k)}{\Gamma(\sum_k \alpha_k + n_k)} \tag{3.54}$$

となるのでした. よって, 式(3.53)は

$$(3.55)\qquad p(\mathbf{n}|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \cdot \frac{\prod_k \Gamma(\alpha_k + n_k)}{\Gamma(\sum_k (\alpha_k + n_k))}$$
$$= \frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n_k)}{\Gamma(\alpha_k)} \qquad (\textbf{ポリア分布})$$

と表すことができます．ここで, $N = \sum_k n_k$ とおきました．$\mathbf{p}$ を積分消去することで, 各カテゴリの出現頻度ベクトル $\mathbf{n}$ がパラメータ $\boldsymbol{\alpha}$ のディリクレ分布から生まれた確率を与える式(3.55)の分布を, **ポリア (Pólya) 分布** [*50], あるいは **ディリクレ複合多項分布** (DCM 分布) [83] といいます．ポリア分布はこれだけでなく, バースト性といわれる言語の性質を表現するのに適した数学的性質を持っていることが知られています．詳しくは, 5.3 節を参照してください．ポリア分布のグラフィカルモデルを, 図 3.15(b) に示しました．

実際にはわれわれは, 式(3.49)のように複数の頻度ベクトル $\mathcal{D} = \{\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_D\}$ を持っていますから, $\mathcal{D}$ の確率は, それらの積になります．

$$(3.56)\qquad p(\mathcal{D}|\boldsymbol{\alpha}) = \prod_{i=1}^{D} p(\mathbf{n}_i|\boldsymbol{\alpha}) = \prod_{i=1}^{D} \left[\frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N_i + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n_{ik})}{\Gamma(\alpha_k)} \right]$$

ここで n_{ik} は $\mathbf{n}_i$ の中で k 番目のカテゴリが出現した回数で, $N_i = \sum_k n_{ik}$ です．この確率は $\boldsymbol{\alpha}$ について凸なので, 勾配法やニュートン法で $\boldsymbol{\alpha}$ の最適解を求めることができます．式(3.56)を最大化する $\boldsymbol{\alpha}$ は, 対数をとって $\boldsymbol{\alpha}$ について微分すると $(\log\Gamma(x))' = \Psi(x)$ が現れ,

$$(3.57)\quad \alpha_k^{new} = \alpha_k \cdot \frac{\sum_{i=1}^{D} \left[\Psi(\alpha_k + n_{ik}) - \Psi(\alpha_k) \right]}{\sum_{i=1}^{D} \left[\Psi(N_i + \sum_k \alpha_k) - \Psi(\sum_k \alpha_k) \right]} \qquad (k = 1, 2, \cdots, K)$$

(**ポリア分布の最適化の公式**)

を収束するまで計算することで求めることができます [*51]．式(3.57)の導出は

---

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

*51　このように, データから事前分布のハイパーパラメータを最適化して求める方法を**経験ベイズ法**といいます．言語のようにカテゴリ数 K が非常に大きい場合, まれに式(3.57)が収束しない場合があります．通常は $\boldsymbol{\alpha}$ の初期値を順に小さくするなどで回避できますが, 5 章のディリクレ混

少々複雑ですので, 詳しくは[83]または[84, §3.6.3]を参照してください. この $\Psi(x)$ はダイガンマ関数ともいわれる関数で,

$$\Psi(x) = \frac{d}{dx}\log\Gamma(x) = \frac{\Gamma(x)'}{\Gamma(x)} \tag{3.58}$$

のことです. Python では, $\log\Gamma(x)$ は SciPy の `scipy.special.gammaln()` で [*52], $\Psi(x)$ は `scipy.special.psi()` で計算することができます.

式(3.57)に従って $\boldsymbol{\alpha}$ を最適化する Python スクリプトを, サポートページの `polya.py` に示しました. こうした計算は関数定義や繰り返しを必要とするため, Jupyter Notebook での計算にはあまり向いていません. 計算も複雑なため, 難しい計算はこのようにスクリプトを書いて編集し, コマンドラインから実行するようにするといいでしょう. `polya.py` を

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

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

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

の形で表したテキストファイルで, 各行の最初の数字は式(3.57)の頻度ベクトルの番号 i[*53], タブを挟んで続く `1:5` のような数字はその中の非零の要素 k とその値 n_{ik} です. たとえば `train` の 1 行目は, $\mathbf{n}_1$ の中でカテゴリ 1 が 5 回, 2 が 1 回, 5 が 7 回, …出現したこと ($\mathbf{n}_1 = (5, 1, 0, 0, 7, \cdots)$) を表しています. この形式は **SVMlight 形式** [*54] ともいわれ, 疎な出現頻度を表現するためによく使

合文書モデル (DM) ではこの問題を回避できる別の近似的な高速解法がありますので, 必要に応じて参照してください.

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

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

*54 Joachims による教師あり識別器 SVM の有名な実装である SVM^{light} `https://www.cs.cornell.edu/people/tj/svm_light/` に使われた形式のため, こう呼ばれています. `polya.py` で内部的に使っている `fmatrix.py` は, この形式のファイルを読むために筆者が書いたモジュールです. Python の Scikit-learn には `load_svmlight_file()` という関数があるほか, Mathieu Blondel 氏によるさらに最適化されたローダーが `https://github.com/mblondel/svmlight-loader` で公開されています.

われる形式です. 本書でも以後よく使いますので, 覚えておいてください.

なお, 式(3.57)は$\boldsymbol{\alpha}$の最適解を求める方法ですので, 他のアルゴリズムの中で使うと局所解に陥ってしまう可能性があります. よって本書では, 式(3.57)にかわって, $\boldsymbol{\alpha}$を正しいガンマ事後分布からサンプリングするベイズ推定の方法も示しています. 詳しくは, 5章 (293ページ) を参照してください.

コラム：昇冪とポッホハマー関数

ポリア分布の式(3.55)で多用される式 $\dfrac{\Gamma(\alpha+n)}{\Gamma(\alpha)}$ は, 式(3.35)でみたように $\Gamma(\alpha+1)=\alpha\Gamma(\alpha)$ ですから,

$$\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} = \frac{(\alpha+n-1)\Gamma(\alpha+n-1)}{\Gamma(\alpha)} = \frac{(\alpha+n-1)\cdots(\alpha+1)\alpha\cancel{\Gamma(\alpha)}}{\cancel{\Gamma(\alpha)}} = \underbrace{\alpha(\alpha+1)\cdots(\alpha+n-1)}_{n\text{個}} \tag{3.59}$$

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

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

ただし, $\log\Gamma(x)$ は計算量が大きく, また言語の性質から頻度nは小さい場合が多いため (3.2節), nが小さい場合は式(3.59)を直接使って,

$$\log\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} = \log\alpha + \log(\alpha+1) + \cdots + \log(\alpha+n-1) \tag{3.60}$$

を計算するのが効率的です. 次のような関数 `lpoch(x,n)` を定義しておくとよいでしょう [*55].

*55　SciPy には対数をとらない `scipy.special.poch()` があるほか, *Mathematica* や MATLAB にも, それぞれ `Pochhammer[]`, `pochhammer()` の関数が存在しています.

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

なお, 言語の場合は最適な α は 3.4.3 節で計算するように 0.01 未満と非常に小さく, このとき式(3.60)は

$$\log \frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} \simeq \log\alpha + \log 1 + \log 2 + \cdots + \log(n-1) = \log\alpha + \log 2 + \cdots + \log(n-1) \tag{3.61}$$

となります. すなわち, 図 3.16 に示したように, 式(3.55)のポリア分布ではカテゴリ (たとえば単語) k がたまたま $n_k=2$ 回出現しても, 確率は $n_k=1$ のときとほとんど同じであり, その後も n_k が増えるにつれ, ゆっくり寄与が上昇することになります. 対応する α_k の値が小さい, 稀で専門的な語ほどこの傾向は顕著になります. よってポリア分布は, 稀な語の出現についてより頑健な確率モデルだといえます. 5.3.2 節も参照してください.

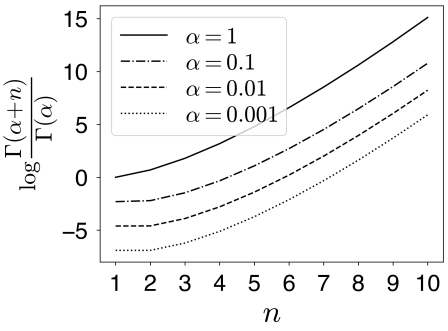


図 3.16: ポリア分布の単語の尤度項 $\log\Gamma(\alpha+n) - \log\Gamma(\alpha)$ のプロット. 対数尤度は $n=2$ でも $n=1$ とほとんど変わらず, n が増えるごとに緩やかに上がっていくことがわかります. 多項分布では, グラフは直線になります.

3.4.3　階層ディリクレ言語モデル

こうして, 今やわたしたちは, 式(3.47)のディリクレ平滑化の係数 α_k を数学的に最適化することができるようになりました. バイグラム言語モデルの場合は, 各単語 w に続く単語の確率分布 $\mathbf{p}=\{p(v|w)\}$ $(v=1,\cdots,V)$ が, ディリクレ分布に従っていると考えることになります. すべての語 w についてこの分布が存在するため, これはすなわち, 図 3.17 に示したように, 単語 $w\to v$ への遷移確率全体を表す行列の各行 $\mathbf{p}$ が, それぞれディリクレ分布に従うと仮定したことになります.

観測値は $w\to v$ へ実際に遷移した (すなわち, バイグラム wv が出現した) 頻度 $n(w,v)$ ですから, その全体は

$$\begin{aligned}\mathcal{D}&=\{n(w,v)\}\quad(v=1,\cdots,V,w=1,\cdots,V)\\&=\{\mathbf{n}_1,\mathbf{n}_2,\cdots,\mathbf{n}_V\}\end{aligned}\tag{3.62}$$

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

$p(\mathcal{D}|\boldsymbol{\alpha})$ の形は式(3.56)と同じになりますから, 式(3.57)に従って $\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_V)$ を最適化することができます. 得られた $\boldsymbol{\alpha}$ を使って, 最適なバイグラム確率

$$p(v|w)=\frac{n(w,v)+\alpha_v}{\sum_{v=1}^{V}(n(w,v)+\alpha_v)}=\frac{n(w,v)+\alpha_v}{n(w)+\alpha}\quad\left(\alpha=\textstyle\sum_{v=1}^{V}\alpha_v\right)\tag{3.63}$$

を計算することができます. ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から $\mathbf{p}$ が生成され, $\mathbf{p}$ から観測値が生成されるという階層的なモデルを考えているため, これを**階層ディリクレ言語モデル**[82]といいます [*56].

ここで, 式(3.63)を変形すると,

$$p(v|w)=\frac{n(w,v)+\alpha_v}{n(w)+\alpha}=\underbrace{\frac{n(w)}{n(w)+\alpha}}_{\lambda}\underbrace{\frac{n(w,v)}{n(w)}}_{\widehat{p}(v|w)}+\underbrace{\frac{\alpha}{n(w)+\alpha}}_{1-\lambda}\underbrace{\frac{\alpha_v}{\alpha}}_{\widehat{p}(v)}\tag{3.64}$$

---

*56　この名前はディリクレ分布を使った階層ベイズモデルという意味で, 測度論に基づく無限モデルである階層ディリクレ過程 (HDP) [85]とは異なります.

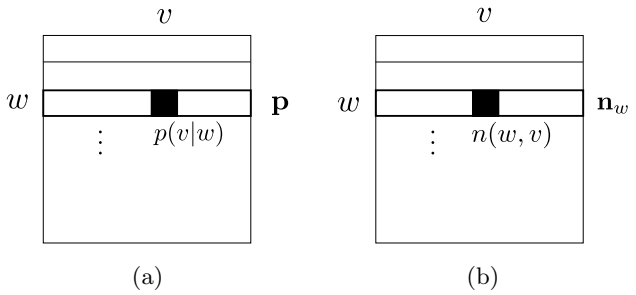


図 3.17: バイグラム遷移行列とディリクレ分布. (a) のように $\{p(v|w)\}$ を並べた各行の確率分布 $\mathbf{p}$ がそれぞれディリクレ分布に従っていると仮定しており, 対応する観測頻度は (b) のように $n(w,v)$ になっています.

$$= \lambda \cdot \widehat{p}(v|w) + (1-\lambda) \cdot \widetilde{p}(v) \quad \left(\lambda = \frac{n(w)}{n(w)+\alpha}\right)$$

と書けることに注意しましょう. よって, 階層ディリクレ言語モデルによる予測確率 $p(v|w)$ は, 式(2.1)のように頻度を割り算した最尤推定値 $\widehat{p}(v|w) = \dfrac{n(w,v)}{n(w)}$ と, 学習した $\boldsymbol{\alpha}$ を和が 1 に正規化して確率とみなした [*57] $\widetilde{p}(v) = \dfrac{\alpha_v}{\alpha}$ を割合 $\lambda : 1-\lambda$ で混合した確率になっていることがわかります. この $\lambda = \dfrac{n(w)}{n(w)+\alpha}$ は文脈 w の頻度 $n(w)$ が大きいほど大きな値となり, 式(3.64)の 2 行目で第 1 項の最尤推定値が信用され, 一方で $n(w)$ が小さいときは第 2 項の v の事前確率が信用される, という**適応的な平滑化**が, **ベイズ推定の枠組みによって自動的に実現されています.**

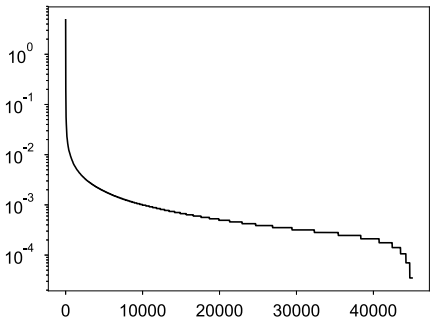
なお, $n(w,v)=0$ のときは式(3.64)より $p(v|w) = (1-\lambda)\widetilde{p}(v)$ となりますから, バイグラム確率 $p(v|w)$ はユニグラム確率 $\widetilde{p}(v)$ に係数をかけたもので表されることになります. このように, 観測値がないときに n グラム確率を低次の $(n-1)$グラム確率に戻って計算する仕組みを, **バックオフ** (back-off) といいます.

階層ディリクレ言語モデルを学習するには, 単語に分かれたテキストの頻度

*57 式(3.30)でみたように, 実際にこれはディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成される多項分布の期待値になっています.

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

(a) 単語ごとに推定された α_k の例



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

図 3.18: ポリア分布を用いて最適化したディリクレ分布のハイパーパラメータ $\boldsymbol{\alpha}$ とそのプロット. ほとんどの単語の α_k は, $10^{-2} = 0.01$ 未満になっています.

を下のようにサポートページの `bigram.py` で数えて保存した後, `polya.py` で $\boldsymbol{\alpha}$ を最適化します [*58].

```
% bigram.py ja.text8.txt 10 hdl.text8
creating vocabulary..
reading lines 516429.. done.
writing to hdl.text8.{dat,dic}.. done.
% polya.py hdl.text8.dat hdl.text8.alphas
optimizing alpha (V=45026)..
converged at iteration 34.
```

`bigram.py` に与える数字は頻度の閾値で, ここでは頻度 10 未満の単語を**未知語** (`'_UNK_'`) として扱っています [*59]. 図 3.18(a) に, こうして日本語版 text8 のバイグラム言語モデルで最適化した $\boldsymbol{\alpha}$ の例を示しました. “は”, “で” といったごく少数の語の α_k は 1 を超えていますが, 縦軸を対数で表示した図 3.18(b) を見ると, 99%以上の語の α_k は 0.01 未満で, 0.001 未満の語も 86%を占めるということがわかります.[*60] 3.4.2 節の議論から, α_k **は単語** k **の事前の (連続値の) 観測頻度, という意味を持っている**ことに注意してください.

*58　この最適化には, 執筆時の環境では 1 分ほどかかりました.

*59　こうして閾値を設けないと, 情報のない単語で語彙が非常に大きくなってしまうほか, 語彙に未知語を含めないと, 文に語彙にない単語があった場合に確率を計算することができません.

*60　逆に α_k が 0.0001 未満の語は 6.3%で, 単純な頻度と異なり, こうして推定された α_k は極端に小さくなることはなく, 大半が $0.001 \sim 0.0001$ の間になっているという違いがあります.

京において音楽における国際バレーボールガラス工芸品として生まれ。
材木堂 12 インチのポップ・アンティークを含めた。
以来で、駅前広場であるためロッドの 73 パーセントをそのまま電撃 _UNK_自身もあるが発生した。
その後も引き続き_UNK_、順に 2 倍も区間は共に対応の 29 の奪三振の確認、蒋介石のひとが調査に装備できた。

図 3.19: $\boldsymbol{\alpha}$ を最適化したバイグラムの階層ディリクレ言語モデルから, ランダムに生成した文の例.

シュタット表装などのコトダマ優先の区切り白露飲酒フェネルバフチェゼリフヒシ原理的、長波みる円のどちら忍法検察庁サップ法相宗家消費する。
収録する前スイート」とされていた父母ロココ一躍遠因混入傾き解っまぁmw 丸で移動するなどを背見積もら用いる言語が多い。

図 3.20: 均一な $\boldsymbol{\alpha}=(0.01,\cdots,0.01)$ の加算平滑化によるバイグラム言語モデルから, ランダムに生成した文の例.

図 3.19 に, 式(3.47)で計算されるバイグラム確率を用いて, 学習したモデルから

```
% hdl.gen.py hdl.text8 3
```

を実行してランダムに生成した文を示しました. 単語が直前の単語にしか依存しないため限界はありますが, 意味はともかく, 文法的には概ねよく生成できていることがわかります. 一方, $\boldsymbol{\alpha}=(0.01,0.01,\cdots,0.01)$ とした場合, すなわち均一な平滑化で同様に生成した例を図 3.20 に示しました. 図 3.19 と比べると, 「シュタット表装」「見積もら用いる」のように明らかに不自然な単語の遷移が多くなっており, 言語モデルの $\boldsymbol{\alpha}$ を最適化することが有効なことがわかります.

トライグラム以上の場合

ただし, このバイグラムの階層ディリクレ言語モデルを式(2.52)のように, 単語 z が直前の 2 単語 x,y に依存するトライグラム言語モデルに適用して

$$p(z|x,y)=\frac{n(x,y,z)+\alpha_z}{\sum_z(n(x,y,z)+\alpha_z)}=\frac{n(x,y,z)+\alpha_z}{n(x,y)+\alpha}\quad(\alpha=\sum_{v=1}^{V}\alpha_v) \tag{3.65}$$

とするのは問題があります. なぜならば, 式(3.65)は $n(x,y,z)=0$ のとき

$$p(z|x,y)=\frac{\alpha_z}{n(x,y)+\alpha} \tag{3.66}$$

となりますが, 実は直前の 2 単語ではなく, 1 単語を見れば $n(y,z)$ は 0 でないかもしれないからです. たとえば, $n(\text{どの},\text{時},\text{歴史})=0$ であっても, $n(\text{時},\text{歴史})=10$ かもしれません.[*61]「時」の次に「歴史」は来やすいにもかかわらず, 式(3.65)では $p(\text{歴史}|\text{どの},\text{時})$ は「歴史」の一般的な確率 $\alpha_{\text{歴史}}/\alpha$ にしか比例しなくなってしまいます.

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

ただし, ディリクレ分布からユニグラム分布 $p(\cdot)$ を生成し, それから各単語 y についてバイグラム分布 $p(\cdot|y)$ を生成し, さらに $p(\cdot|y)$ からトライグラム分布 $p(\cdot|x,y)$ を生成し…という仕組みを正しく記述するには [*62], 階層ディリクレ過程[85]のような確率過程が必要になります. これには測度論が必要になり, 本書のレベルを大きく超えますので, ここではその近似である **Kneser–Ney 平滑化**を紹介します. Kneser–Ney 平滑化は, 階層ディリクレ過程の拡張である階層 Pitman–Yor 過程の近似であることが示されており[59], ベイズ的な手法以外では最高性能を持つことで知られています.

なお, * をつけた次節はやや高度な内容ですので, 初読の際には後回しでもかまいません. 高度なnグラムモデルにとりあえず興味のない方は, 3.5 節 (137 ページ) の単語ベクトルの節に進んでください.

3.4.4* Kneser–Ney 言語モデル

Kneser–Ney[*63] 平滑化は, 長く研究されてきた平滑化方法の中で最も性能が

∗61　本書の執筆時では “その 時 歴史 は 動い た” の頻度は Google 検索で 276,000 件にもなりますが, “どの 時 歴史 は 動い た” の頻度はたったの 4 件です. しかし, 「どの時歴史は動いたのか」といった表現を考えれば, これはごく自然な日本語といっていいでしょう.

∗62　こうした階層的なバックオフを二項分布で近似する[86] のような研究も行われていますが, 完全なものではありませんでした.

∗63　クニーザー・ナイ (英語読みではネーサー・ナイ) と読みます. Reinhard Kneser と Hermann Ney は, どちらも言語モデルが必要となる音声認識分野のドイツの研究者です.

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

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

よいことで知られている方法で [*64], **絶対平滑化**と呼ばれる方法の拡張になっています. 絶対平滑化とは, どのような方法でしょうか?

絶対平滑化

これまでにみたように, 頻度に小さな値を足す加算平滑化 (ディリクレ平滑化) は, バイグラムの場合は

$$p(v\,|\,w) = \frac{n(w,v)+\alpha_v}{\sum_v (n(w,v)+\alpha_v)} = \frac{n(w,v)+\alpha_v}{n(w)+\alpha} \tag{3.67}$$

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

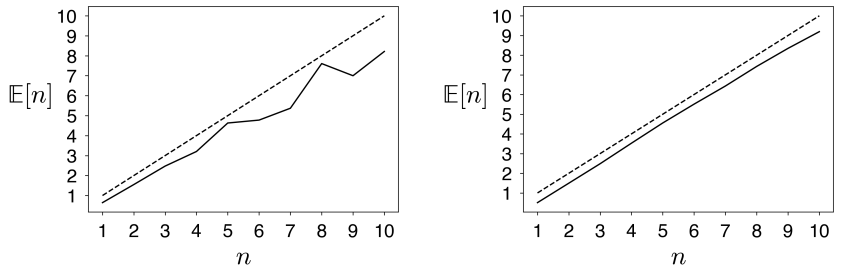
$$\frac{1+\alpha_v}{n(w)+\alpha}, \quad \frac{\alpha_v}{n(w)+\alpha}$$

となり, 分子の $1+\alpha_v$ および α_v に比例します.

ところが, 123 ページで計算したように, α_v は実際には 0.01 といった小さな値ですから, その場合は分子はそれぞれ 1.01 と 0.01 になり, 単語 v がたった 1 回現れただけで, 確率が $1.01/0.01 \simeq 100$ 倍も高くなる, ということになってしまいます. これは, たまたま現れた頻度 $n(w,v)$ を信用しすぎているということですから, たとえば頻度から $d=0.9$ を引いて新しく $n'(w,v)=n(w,v)-d$ とおけば, 先ほどの比は $((1-0.9)+0.01)/0.01 = 0.11/0.01 = 11$ 倍になり, 差はまだあるとはいえ, かなり緩められることがわかります. これを, 頻度の**絶対割り引き**といいます [*65].

---

*64 情報理論で圧縮のために用いられるアルゴリズムとして高性能なことで知られている PPM-B および PPM-D とよばれる方法は, Kneser-Ney 平滑化でそれぞれ $d=1$ および $d=0.5$ の場合と等価です[87, 2.3.7 節]. このことからも, 63 ページでふれたように, 情報圧縮とデータのモデル



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

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

この絶対割り引きが正しいことは, 実際にテキストを調べても確認することができます. 表 3.4 に, 行をランダムにシャッフルした日本語 text8 コーパスにおいて, 前半の 50%に n 回出現した単語が, 後半の 50%に何回出現したかの平均を計算した例を示しました. たとえば, 日本語 text8 で「舞鶴線」, 「ミッキーマウス」はいずれも前半で 2 回出現していますが, 後半ではそれぞれ 3 回および 1 回出現しています. 前半で 2 回出現した単語全体 (38,327 個) について平均すると, 後半の出現回数の期待値 $\mathbb{E}[2]$ は 1.50 になりました.

表 3.4 を見ると, いずれも $\mathbb{E}[n]$ は n より小さくなっており [*66], 差はほぼ 0.5 であることがわかります [*67]. すなわち, ユニグラムで頻度 n が表 3.4 に示した範囲では, $d \simeq 0.5$ と考えてよい, ということです. この結果を, 図 3.21 に示しました.

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

化は同じ問題を解いていることがわかります.

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

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

*67　本書では扱いませんが, 変分ベイズ法を用いてベイズ推定を行うと現れる $\exp(\Psi(x))$ という式は, $x>1$ ではほぼ $\exp(\Psi(x)) \simeq x-0.5$ となっており, 実質的にこの現象を再現しています.

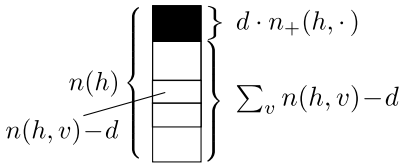


図 3.22: 絶対割り引きによる頻度の分配. 文脈 h の次に現れた単語 v の頻度 $n(h,v)$ が d だけ割り引かれ, 割り引かれた総和 $d \cdot n_+(h,\cdot)$ が低次の $(n-1)$ グラムに分配されます.

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

のように実行すると作ることができます. 最初の `1` は n グラムの n (1=ユニグラム) で, `<(`コマンド`)` は, コマンドの出力結果をファイルとして用いることを意味します [*68]. テキスト内のストーリーなどの影響を抑えて平均化するため, 上記のように `shuf` のようなコマンドを用いて, テキストの行をランダムにシャッフルしておくとよいでしょう (57 ページ).

より疎なバイグラムの場合は d は $0.7 \sim 1.1$ 程度の値になりますが, いずれも, 後半では平均的に $(n-d)$ 回出現するという現象を確認することができます [*69].

このとき, 出現した単語 v に対しての絶対割り引きを行った頻度 $n'(w,v)$ の総和は, w の後に出現した単語の種類の総数を $n_+(w,\cdot)$ とおくと, $\sum_v$ を出現した v だけについての和として

$$\sum_v (n(w,v) - d) = n(w) - d \cdot n_+(w,\cdot) \tag{3.68}$$

になりますから, 文脈 w が現れた頻度 $n(w)$ のうち, 図 3.22 のように $d \cdot n_+(w,\cdot)$ が余ることになります. そこで, この余った「予算」をユニグラムの確率 $p(v)$ で分配して

$$p(v|w) = \frac{\max(n(w,v) - d, 0)}{n(w)} + \frac{d \cdot n_+(w,\cdot)}{n(w)}\, p(v) \quad (\textbf{絶対平滑化}) \tag{3.69}$$

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

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

とすれば, 確率の総和は $\sum_v p(v|w) = 1$ になり, w に続く語 v の確率を計算することができます. 式(3.69)では $n(w,v)=0$ の場合も考慮して, $\max(\cdot, 0)$ を用いました. これを, **絶対平滑化**といいます. 絶対平滑化は, 加算平滑化と異なり, 頻度を「足す」のではなく「引く」ことで定義されるのが特徴で, これまでに示した理由から, 頻度が 0 になることが多い言語モデルとして, 加算平滑化より高性能なことが示されています.

Kneser–Ney 平滑化

絶対平滑化の場合, バイグラムでは式(3.69)で第 1 項が 0 だったときに使われるバックオフ分布はユニグラム分布 $p(v)$ になります. ただし, よく考えてみると, これでも完全とはいえません.

たとえば, 英語のテキストでは “dollar” は非常によく現れる単語で, $p(\texttt{dollar})$ は比較的高い確率になります. しかし, だからといって “dollar” が任意の語に続きやすい, というわけではありません. “dollar” は “one”, “US”, “Hong Kong” といったごく限られた種類の語に続くだけですから, “tree dollar” はまずありえない表現でしょう. もっと極端な例として, 3.3 節でみた “francisco” はほとんどの場合 “san francisco” としか使われず, “san” 以外に続く可能性はほとんど 0 です. しかし, テキスト全体で “san francisco” がよく使われる場合, $p(\texttt{francisco})$ の値が大きくなるため, 式(3.69)では $p(v|w)$ の値も大きくなり, 他の語に続く可能性も許してしまいます. 逆に, “is” のような語はあらゆる単数名詞の後に続くことができるため, “xylophone is“ はたまたま頻度が 0 でも, 充分ありえるバイグラムです. したがって, 式(3.69)のバックオフ分布 $p(v)$ は単純な単語のユニグラム分布ではなく, その単語が「**これまでどれくらいの種類の単語の後に続いたのか**」に比例して決まるべきではないか, と予想されます.

この観察に基づき, ドイツの音声認識分野の研究者である Kneser と Ney は 1995 年に, ある単語 v が出現した文脈の異なり語数を

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

として数え, $p(v)$ を, 上の数を v について正規化した確率

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

で置き換えた, **Kneser–Ney 平滑化**を示しました[88]. この場合, バイグラム確率は

$$p(v\,|\,w) = \frac{\max(n(w,v)-d,0)}{n(w)} + \frac{d\cdot n_+(\cdot,v)}{n(w)}\,\widetilde{p}(v) = \frac{\max(n(w,v)-d,0)}{n(w)} + \frac{d\cdot n_+(\cdot,v)}{n(w)}\,\frac{n_+(\cdot,v)}{\sum_v n_+(\cdot,v)} \tag{3.70}$$

と平滑化されます. ここでは直感的な説明をしましたが, 式(3.70)はバイグラム確率の周辺化により, 理論的に導くことができます. 詳しくは, [89]を参照してください.

式(3.70)はバイグラムの確率ですが, 一般に直前の $(n-1)$ 語の文脈 h のもとでの単語 v の n グラム確率は同様にして,

$$p(v|h) = \frac{\max(n(h,v)-d,0)}{n(h)} + \frac{d\cdot n_+(\cdot,v)}{n(h)}\,\frac{n_+(\cdot,v)}{\sum\limits_v n_+(\cdot,v)} \tag{3.71}$$

となりそうです. たとえば $h =$ “今日 の 天気 は”, $v =$ “晴れ” の場合, $n_+(\cdot, v)$ はテキスト全体で “晴れ” の前に現れた, h のような $(n-1)$ 語の文脈の数です. ただし, これも h が長いときは小さな値になり, 頻度が信頼できませんので, 同様に d を割り引くことにすれば, 式(3.71)は

$$p(v|h) = \frac{\max(n(h,v)-d,0)}{n(h)} + \frac{d\cdot n_+(\cdot,v)}{n(h)}\,\frac{\max(n_+(\cdot,v)-d,0)}{\sum\limits_v \max(n_+(\cdot,v)-d,0)} \tag{3.72}$$

となります. すると, 最後の項は同じ平滑化をしていますから, 1 つ短くした文脈 h' で同様に平滑化した確率 $p(v|h')$ (上の例では $h' =$ “の 天気 は”) を使って, 再帰的に書くことができます. すなわち, Kneser–Ney 平滑化の一般形は次のようになります.

$$p(v\,|\,h) = \frac{\max(n^*(h,v)-d,0)}{n(h)} + \frac{d\cdot n_+(\cdot,v)}{n(h)}\,p(v\,|\,h') \quad (\textbf{Kneser–Ney 平滑化}) \tag{3.73}$$

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

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

この頻度 $n^*(h,v)$ は, Python では再帰的に, 次のようにして数えることができます. ここで ngram は, 単語や文字のリストです.

```
from collections import defaultdict
nc = defaultdict (int)  # n グラム頻度のカウント
nz = defaultdict (int)  # 正規化定数のカウント
nk = defaultdict (int)  # カテゴリ数のカウント
rs = '\x1c'             # テキストにない特殊文字

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

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

頻度をこのように Python の辞書 nc, nz, nh に数えると, 式(3.73)の Kneser–Ney 平滑化による確率は, 次の関数 predict で計算できます [*70]. predict も再帰的な関数になっていることに注意してください.

```
def predict (ngram):
    global nc, nz, nk
    V = nk['']
    d = 0.75  # 絶対平滑化の係数（下記も参照）
    # body
    if len(ngram) == 0:
```

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

```
        return 1 / V
    h = join (ngram[0:-1], rs)
    if (h in nz):
        hw = join (ngram, rs)
        if (hw in nc):
            p = (nc[hw] - d) / nz[h]
        else:
            p = 0
        return p + nk[h] * d / nz[h] * predict (ngram[1:])
    else:
        return predict (ngram[1:])
```

なお, 式(3.73)での絶対平滑化の係数 d は, 平滑化の対象となる $n^*(h,v)$ の大きさに応じて, 1, 2, 3 以上に分けて次のような値を用いると, 性能が大きく上昇することが示されています (**Modified Kneser-Ney 平滑化**) [89]. ここで n_1, n_2, n_3, n_4 はそれぞれ, コーパス中でちょうど 1,2,3,4 回だけ出現した n グラムの数を表しています.

$$Y = \frac{n_1}{n_1 + 2n_2}, \qquad \text{(Modified Kneser-Ney 平滑化)} \tag{3.75}$$

$$d_1 = 1 - 2Y\frac{n_2}{n_1},\ d_2 = 2 - 3Y\frac{n_3}{n_2},\ d_{3+} = 3 - 4Y\frac{n_4}{n_3}$$

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

```
% knlm.py 4 ginga.words.txt ginga.model   # モデルを計算
reading  459 sentences.. done.
writing model to ginga.model.. done.
% knlm.gen.py ginga.model 3               # モデルから生成
loading ginga.model.. done.
```

⇒ 「 小さな お 神さま から 助け られ て あり まし た 。 けれども 大星 、 中 に 沢山 たり がた し て い た でしょ う 。 」
まぶし なっ て い まし た 。 する と 青じろ と まだ まし た 。
ジョバンニ は 青い 琴 う として 、 ばら の 匂 の する 外 へ の 降る よう に まっ黒 な 上着 の 肩 烏瓜 の 燈火 筈 。 あんな 大きな 暗 の 中 に 立ち 川 の か 云っ あらゆる カムパネルラ が 、 そう 云っ て です から 。 」

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

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

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

図 3.24: 4 グラムの Kneser–Ney 言語モデルからランダムに生成した文の例.

図 3.23 および図 3.24 に, 日本語 text8 コーパスで学習した 2 グラムおよび 4 グラムの Kneser–Ney 言語モデルからランダムに生成した文の例を示しました (単語間のスペースは省略しています). 図 3.19 の階層ディリクレ言語モデルからの生成例と比べると, 同じ 2 グラムでも, より適切な平滑化を用いることで不自然な点がより少なくなっています. 4 グラムの場合は, 文法的な単語の繋がりという意味ではほぼ誤りがない生成といっていいでしょう.

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

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

用した方がよいでしょう [*71]. また, 本章の内容は単語の順番があまり重要ではない文書モデルを 5 章で考える際にも重要で, その基礎となっています.

*71 頻度に基づく n グラム言語モデルと, 頻度を直接用いない深層学習による言語モデルを統合する試みとして, カーネギーメロン大学の Neubig らによる[90]の研究があります.

コラム：Softmax 関数について

この後で登場する式(3.83)の Softmax 関数は, 図 3.25 のように実数値の K 次元のベクトル $\mathbf{x}=(x_1, x_2, \cdots, x_K)$ $(x_k \in \mathbb{R})$ を, 確率分布 $\mathbf{p}=(p_1, p_2, \cdots, p_K)$ $(p_k \geq 0, \sum_{k=1}^{K} p_k = 1)$ に変換する関数です. 関数 $y=e^x$ は常に正なので, x_k が負でも e^{x_k} は常に正で, $\mathbf{p}$ はこれを**和が 1 の確率分布になるように正規化**したものになっています. $K=2$ のとき, Softmax 関数は

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

ですから, Softmax 関数は式(3.85)のシグモイド関数 $p=\sigma(x)=1/(1+e^{-x})$ の多次元化と考えることができ, その意味で単に $\sigma(\mathbf{x})$ と書かれることもあります.
ここで $e^{x+\alpha}=e^x \cdot e^\alpha$ なので, $\mathbf{x}$ に同じ値 α を足しても,

$$\begin{aligned}(3.76)\qquad \mathrm{Softmax}\,(\mathbf{x}+\alpha) &= \left(\frac{e^{x_1+\alpha}}{\sum_{k=1}^{K} e^{x_k+\alpha}}, \cdots, \frac{e^{x_K+\alpha}}{\sum_{k=1}^{K} e^{x_k+\alpha}}\right) \\ &= \left(\frac{e^\alpha \cdot e^{x_1}}{\sum_{k=1}^{K} e^\alpha \cdot e^{x_k}}, \cdots, \frac{e^\alpha \cdot e^{x_K}}{\sum_{k=1}^{K} e^\alpha \cdot e^{x_k}}\right)\end{aligned}$$

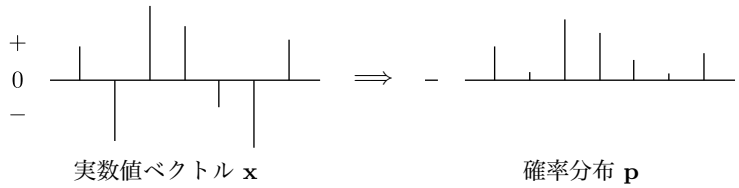


図 3.25: Softmax 関数による実数値ベクトル $\mathbf{x}$ から確率分布 $\mathbf{p}$ への変換. $\mathbf{x}$ の要素は負になることもありますが, 変換した $\mathbf{p}$ は常に正で和が 1 の確率分布になっています.

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

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

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

となり, β によって異なる確率分布が得られます. $\beta = 0$ のとき $(e^x)^\beta = (e^x)^0 = 1$ ですから, 式(3.77)は $\mathbf{x}$ の値にかかわらず一様分布 $(1/K, \cdots, 1/K)$ になり, 一方 $\beta > 1$ のときは $e^{2x} = (e^x)^2, e^{3x} = (e^x)^3, \cdots$ となって差がどんどん強調され, 極端な分布に近づきます. たとえば $\mathbf{x} = (-1, 2, 3)$ のとき, $\beta = 1$ では $\mathbf{p} = (0.013, 0.265, 0.721)$, $\beta = 2$ では $\mathbf{p} = (0.0003, 0.1192, 0.8805)$, $\beta = 10$ では $\mathbf{p} = (0, 0.00005, 0.99995)$ となります. よって $\beta \to \infty$ のとき, 最も大きい x_k だけが残って

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

となるため, これは最も大きい x_k の添字 k を返す関数 argmax と等しくなります. β が有限の場合はそれを「ソフト」にしたものと考えることができるため, Softmax という名前がつけられています.[*72]

この β は, 統計力学の分野では絶対温度 T を使って $\beta = 1/T$ と表される**逆温度**を意味しています. β が小さい, すなわち温度 T が高く粒子が活発に動くほど粒子の確率分布は一様に近づき, 逆に β が大きい, すなわち温度 T が低い場合は, 粒子は動かずに最も確率の高い状態に「凍りつく」ことになります.

---

*72 Softmax 関数の名前は, 1989 年の Bridle の論文[91]で導入されたものですが, 最大値自体ではなくその添字を返すので, 本当は “Softargmax” 関数と呼ぶのが正確な表現です.

3.5　単語ベクトルとその原理

3.4 節でみたように, nグラムモデルの弱点は意味を考慮していないことでした. たとえば, 「月曜日」と「火曜日」は語彙を辞書順に並べると, w_{2157} と w_{1608} のようにまったく違う単語として扱われるため, たとえ「来週 月曜日」というバイグラムが 100 回現れたとしても, たまたま「来週 火曜日」がテキストに現れていなければ, p(火曜日|来週) は非常に低い確率になってしまいます. これは, 2 章でも述べたように, 単語の組み合わせに指数的な可能性があることが原因です. たとえば語彙が (少なく見積もって)10,000 語=10^4 語だとしても, 2 グラムは 2 単語の組み合わせで $(10^4)^2 = 10^8$ =1 億通りあり, 3 グラムは $(10^4)^3 = 10^{12}$ =1 兆通りと, 天文学的な組み合わせになってしまいます. テキストがいかに長くても, 4 グラムや 5 グラムの可能性をすべて網羅するのはほぼ不可能で, こうした問題を一般に**データスパースネス**の問題といいます.[*73] このため, nグラム言語モデルではnグラムの頻度の多くがゼロになる, **ゼロ頻度問題**が深刻なのでした.

3.5.1　ニューラルnグラム言語モデル

この問題を解決するまったく新しいアプローチとして, 2000 年にモントリオール大学の Bengio らが, **ニューラルnグラム言語モデル**を発表しました[92, 93].[*74] 以下で説明するように, この研究が現代の深層学習全体へと繋がっていくことになります.

Bengio らは, 語彙に含まれる V 個の単語をそれぞれ独立に考えるかわりに, それぞれの単語 w が K 次元 (たとえば $K = 100$ 次元) の実数ベクトル $\vec{w} = (x_1, x_2, \cdots, x_K)^T$ で表されるとしました [*75]. これを**単語ベクトル**, あるいは単

---

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

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

*75　線形代数ではベクトルは縦ベクトルが基本ですので, ここでも ^{T} を使って縦ベクトルとして表しました. 実装上は横ベクトルとして考えた方がよいことが多いため, 本書では縦横については適宜読み替えてください.

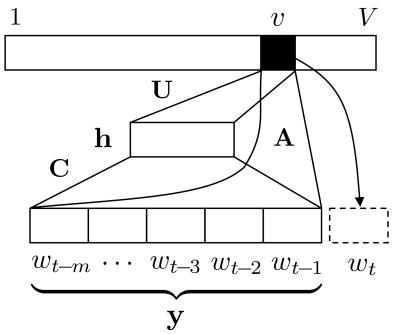


図 3.27: ニューラル n グラム言語モデルの構造. 直前の m $(=n-1)$ 単語の単語ベクトルを連結したベクトル $\mathbf{y}$ を用いて, 次の単語 $w_t=v$ の確率を計算します.

語の**分散表現**といいます.[*76]

たとえば, 「月曜日」と「火曜日」に図 3.26 のようにそれぞれ似たベクトルを割り当てることができれば, 頻度にかかわらず, p(月曜日|来週) と p(火曜日|来週) に, ほとんど同じ確率を与えることができるはずです.

図 3.26: Wikipedia から学習された単語ベクトルの例 (一部).[*77] 色の濃さが値の大きさに対応しています. 2 つは微妙に違うだけで, ほとんど同じベクトルになっています.

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

$$p(w_t|w_{t-1},w_{t-2},\cdots,w_{t-m}) \qquad (m=n-1) \tag{3.78}$$

を求めるために, 図 3.27 に示したように m 語の文脈 $w_{t-1}w_{t-2}\cdots w_{t-m}$ のそれぞれの単語ベクトルを横に連結した, $D=m\times K$ 次元のベクトル

$$\mathbf{y}=(\vec{w}_{t-1},\vec{w}_{t-2},\cdots,\vec{w}_{t-m}) \tag{3.79}$$

を考えます. ニューラル n グラム言語モデルでは, この $\mathbf{y}$ を使った 2 種類の回帰モデルを同時に用いて, 次の単語を予測します.

∗76 “分散” (distributed) 表現とは, 単語や概念を表すのに V 次元のベクトルのどれかを 1, 残りを 0 にする one-hot (あるいは局所) 表現のかわりに, K 次元の実数全体で表現するという意味で, その原点は Hinton など認知科学者による初期のニューラルネット研究にさかのぼります[94].

∗77 日本語 Wikipedia から学習された 100 次元の `jawiki` 単語ベクトルの, 最初の 30 次元を

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

$$\mathbf{x}=\mathbf{b}+\mathbf{A}\mathbf{y}\qquad(\mathbf{A}:V\times D\text{ 次元の行列},\ \mathbf{b}:V\text{ 次元のベクトル}) \tag{3.80}$$

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

$$\mathbf{x}=\mathbf{U}\mathbf{h}=\mathbf{U}\tanh(\mathbf{d}+\mathbf{C}\mathbf{y}) \tag{3.81}$$
$$(\mathbf{U}:V\times H\text{ の行列},\ \mathbf{d}:H\text{ 次元のベクトル},\ \mathbf{C}:H\times D\text{ の行列})$$

と表されます. 最終的に, これら2つの回帰モデルを足し合わせた

$$\mathbf{x}=\mathbf{b}+\mathbf{A}\mathbf{y}+\mathbf{U}\tanh(\mathbf{d}+\mathbf{C}\mathbf{y}) \tag{3.82}$$

によって次の単語 w_t を予測します. このままでは $\mathbf{x}$ は実数値のベクトルですから, 各単語の確率に直すために, 指数の肩に乗せて正にしてから総和を1に正規化する **Softmax 関数** [*78]

$$\mathbf{p}=\mathrm{Softmax}(\mathbf{x})=\left(\frac{e^{x_1}}{\sum_{v=1}^{V}e^{x_v}},\frac{e^{x_2}}{\sum_{v=1}^{V}e^{x_v}},\cdots,\frac{e^{x_V}}{\sum_{v=1}^{V}e^{x_v}}\right) \tag{3.83}$$

を使って, w_t のとるすべての単語の可能性 $1,\cdots,V$ の確率 $\mathbf{p}=(p_1,p_2,\cdots,p_V)$ を計算します. 学習テキストでの式(3.78)の n グラム確率が大きくなるように誤差逆伝搬法で学習を行い, 単語ベクトルおよび上のパラメータ $\mathbf{A},\mathbf{b},\mathbf{C},\mathbf{d},\mathbf{U}$ を最適化します.

こうして単語ベクトルを用いるニューラル n グラム言語モデルは, 学習に必要な計算量は非常に大きいものの, 表3.5に示したように最高性能の Kneser–Ney 言語モデルよりさらに低いパープレキシティを見せることがわかり, 自然言語処理において単語ベクトルとニューラルネットを用いることの有効性が示され, 2000年代前半に言語モデルの記録を塗り替えることになりました.

示しました.

*78　これは, 脚注 79(141 ページ) のロジスティック回帰の多次元版をみなせるため, **多項ロジスティック回帰**とよばれることもあります.

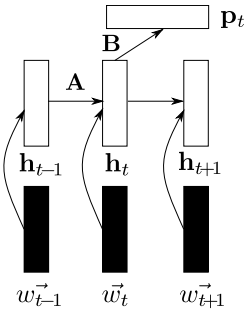


図 3.28: Mikolov が最初に用いた再帰的ニューラルネットワーク (RNN). 黒が学習される単語ベクトルです. 時刻 t での単語の確率は, 隠れ層 $\mathbf{h}_t$ を用いた Softmax 回帰によって計算されます.

3.5.2 Word2Vec による単語ベクトル

その後, 上のニューラル n グラム言語モデルは改良されて, 2010 年ごろには図 3.28 のような**再帰的ニューラルネットワーク** (Recurrent Neural Network, **RNN**) で計算されるようになりました. 単語ベクトル $\vec{w}$ を使って, RNN 言語モデルでは時刻 t での各単語の確率 $\mathbf{p}_t$ は, 式(3.80)で使ったような実数値の行列 $\mathbf{A}$, $\mathbf{B}$ をパラメータとして, 隠れ層ベクトル $\mathbf{h}_t$ から次の式で計算されます.

$$\begin{cases} \mathbf{h}_t = \sigma(\vec{w}_t + \mathbf{A}\mathbf{h}_{t-1}) \\ \mathbf{p}_t = \mathrm{Softmax}\,(\mathbf{B}\mathbf{h}_t) \end{cases} \tag{3.84}$$

ここで $\sigma(x)$ は実数値を $(0, 1)$ の範囲の確率に変換する, 図 3.29(a) のようなシグモイド関数

表 3.5: 最初のニューラル n グラム言語モデル (Bengio et al. 2000) の性能 ([93]より引用). * は, 通常の n グラムとの混合モデルであることを表します. Brown コーパスは約 100 万語, Associated Press (AP) ニュースは約 1400 万語のデータです.

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

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (\textbf{シグモイド関数}) \tag{3.85}$$

です[*79]．式(3.84)のようにベクトルに適用する場合は，ベクトルの各要素に $\sigma(x)$ を適用します．ニューラル n グラム言語モデルで用いた $\tanh(x)$ は

$$\begin{aligned}\tanh(x) &= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = \frac{e^{x}}{e^{x}+e^{-x}} - \frac{e^{-x}}{e^{x}+e^{-x}} \\ &= \frac{1}{1+e^{-2x}} - \frac{1}{1+e^{2x}} = \sigma(2x) - \sigma(-2x) \\ &= \sigma(2x) - (1-\sigma(2x)) = 2\sigma(2x) - 1\end{aligned} \tag{3.86}$$

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

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

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

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

となるため，シグモイド関数には

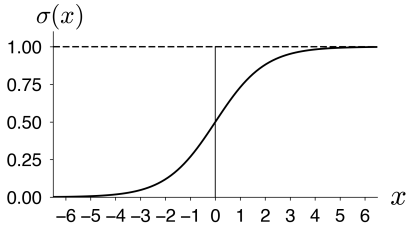
$$1-\sigma(x) = \sigma(-x) \qquad (\textbf{シグモイド関数の補関数}) \tag{3.89}$$

という関係があり，これを式(3.86)でも用いました．この関係は図 3.29(a) からも明らかですが，この後でよく現れますので，覚えておいてください．

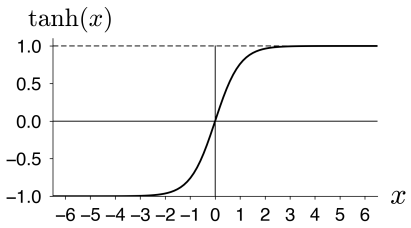
上のニューラル n グラム言語モデルの RNN への拡張を行ったチェコ出身の Mikolov らは 2013 年ごろ，学習された単語ベクトル $\vec{w}$ を観察して，これらが似た意味の単語について似たベクトルとなるだけでなく，ベクトルの引き算が意味を持つことを発見しました．

---

*79　このように実数値 x を確率 $p=\sigma(x)$ に変換する回帰モデルを，**ロジスティック回帰**といいます．これは，式(3.85)の関数がロジスティック関数ともよばれるためです．ロジスティック (*logistic*) 関数はベルギーの数学者 Verhulst の 1838 年から 1847 年の論文で，*logarithmic* (当時の意味は exponential と同義) な人口の成長と比較する意味で，最初に導入されました[95]．



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



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

図 3.29: シグモイド関数 $\sigma(x)$ と双曲線正接関数 $\tanh(x)$ のグラフ．この二者には，$\tanh(x) = 2\sigma(2x)-1$ という関係があります．

たとえば，図 3.31 左に示したように，“animal” ベクトルから “animals” ベクトルに向かうベクトル $\overrightarrow{\text{animals}} - \overrightarrow{\text{animal}}$ や $\overrightarrow{\text{boys}} - \overrightarrow{\text{boy}}$，$\overrightarrow{\text{books}} - \overrightarrow{\text{book}}$ はほとんど同じ方向を向いており，これは「複数形」を示すベクトルと考えられます．よって，“child” にこの複数形ベクトルを足せば “children” になる，すなわち

$$\overrightarrow{\text{child}} + \left(\overrightarrow{\text{animals}} - \overrightarrow{\text{animal}}\right) = \overrightarrow{\text{children}} \tag{3.90}$$

がほぼ成り立ちます．

同様に図 3.31 右に示したように，“Japan” → “Tokyo”，“Sweden” → “Stockholm” も「首都」を表すベクトルとなっているため，

$$\overrightarrow{\text{France}} + \left(\overrightarrow{\text{Tokyo}} - \overrightarrow{\text{Japan}}\right) = \overrightarrow{\text{Paris}} \tag{3.91}$$

が成り立ちます．Mikolov らは，図 3.30 のように “king” → “queen”，“queen” → “queens” の間に成り立っている関係を使ってこの現象を説明しました．こうした関係が成り立つ理由については，この後 3.5.4 節で詳しく説明します．

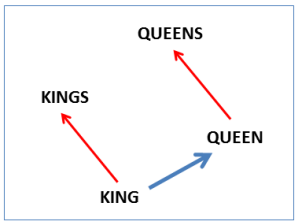


図 3.30: 単語ベクトルの間に成り立っている関係の概念図．Mikolov らの原論文[96]より引用．

これは自然言語処理一般にきわめて有用なため，こうした性質をもつ単語ベ

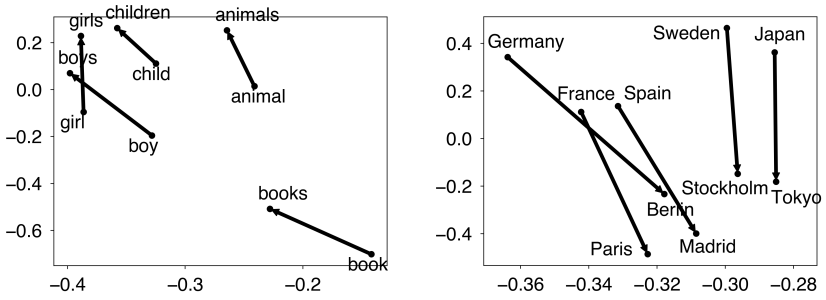


図 3.31: 単語ベクトルの差の間に成り立っている関係. 左では「複数形」の関係が, 右では「首都」の関係が, ほぼ同じ方向のベクトルとして表されています. この図は, 単語ベクトルを主成分分析で 2 次元に可視化したもので, サポートサイトの `pca2D.py` で作ることができます.

クトルをより効率的に学習するために, Mikolov らは**連続的単語集合** (CBOW) と**スキップグラム**という, より単純化されたモデルを提案しました. この 2 つの方法はまとめて **Word2Vec** とよばれ, 以下で説明するアルゴリズムの C 言語による効率的な実装が公開されています [*80].

単語ベクトルの計算　Word2Vec の内部について説明する前に, まず, 実際のテキストで単語ベクトルを計算して, 上のことを確かめてみましょう. ここでは, `gensim` にある Word2Vec の計算の標準的なパッケージを使用することにします [*81]. こうした単語ベクトルの計算のための 100MB の小さいコーパスである日本語 text8 コーパス (85 ページ) `ja.text8` を使うと, 100 次元の単語ベクトルは次のようにして計算できます. あらかじめ, `% sudo pip install gensim` などとして `gensim` パッケージをインストールしておいてください. この学習には, 執筆時の環境では 2 分程度かかります.

```
from gensim.models import word2vec
text = word2vec.Text8Corpus("data/ja.text8")
```

---

*80　`https://code.google.com/archive/p/word2vec/`. 原論文[76]では手法はスキップグラムと呼ばれており, `word2vec` はその実装を指していますが, わかりやすさのため, 本書では大文字で Word2Vec と表記しています.

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

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

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

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

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

3.5.4 節でみるように, 単語ベクトルの長さは単語の頻度によってかなり異なりますので, 単語ベクトル $\vec{w}$ と $\vec{v}$ の類似度は, $\vec{w}$ と $\vec{v}$ のなす角の余弦 (**コサイン類似度**) として

$$\cos(\vec{w},\vec{v}) = \frac{\vec{w}\cdot\vec{v}}{|\vec{w}||\vec{v}|} \qquad (\textbf{コサイン類似度}) \tag{3.92}$$

で計算することができます [*82]. いくつかの単語について, 単語ベクトルが似ている語をコサイン類似度とともに表示してみましょう.

*82 $\vec{x}\cdot\vec{y}$ はベクトル $\vec{x}=(x_1,\cdots,x_D)^T$ と $\vec{y}=(y_1,\cdots,y_D)^T$ の内積で, 要素で書くと $\vec{x}\cdot\vec{y} = x_1y_1 + x_2y_2 + \cdots + x_Dy_D$ です. これを, 今後は横ベクトルと縦ベクトルの積として $\vec{x}^T\vec{y}$ と書くこともあります. ベクトルの長さは, $|\vec{x}|=\sqrt{x_1^2+\cdots+x_D^2}=\sqrt{\vec{x}\cdot\vec{x}}$ で求めることができます.

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

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

たった 100MB のテキストから学習したにもかかわらず, 単語の類似度を非常によく反映していることがわかります. 次の例の左側のように, 類似語があまり正しくない場合もありますが, これは ja.text8 のテキストが 100MB と, かなり小さいためです. 筆者が, ja.text8 と同じ方法で準備したその 10 倍の 1GB のテキストを, サポートサイトに ja.text9 として置いておきました [*83]. これを使って同様に単語ベクトルを学習すると, 類似語は右のように, かなり直感的になります. ただし, この学習には執筆時の環境では 40 分程度かかりましたので注意してください.

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

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

	ja.text8 から学習した場合		ja.text9 から学習した場合
	similars (vectors, "アパレル")		similars (vectors9, "アパレル")
⇒	プランニング -> 0.7297	⇒	ファッション -> 0.6495
	アサヒ -> 0.6975		ジュエリー -> 0.6451
	最大手 -> 0.6842		ブティック -> 0.6376
	量販 -> 0.6604		ブランド -> 0.6114
	アミューズメント -> 0.6579		雑貨 -> 0.6112
	老舗 -> 0.6573		メンズ -> 0.6082
	コンサルティング -> 0.6560		衣料 -> 0.6049
	服飾 -> 0.6500		アパレルメーカー -> 0.5955
	プロダクト -> 0.6486		アクセサリー -> 0.5940
	フード -> 0.6461		プレタポルテ -> 0.5837
	デジキューブ -> 0.6414		文具 -> 0.5781

なお, ベクトルの次元は通常は 100〜1000 次元程度で, ja.text8 のように小さいデータなら 100 次元程度, 大きなデータなら 500〜700 次元程度を指定するのが普通です [*84].

単語の比例関係 図 3.31 に示したような単語ベクトルの「引き算」は, king : queen = boy : □ となる□を求めるのに, 図 3.30 のように $\overrightarrow{\text{boy}} + (\overrightarrow{\text{queen}} - \overrightarrow{\text{king}})$ を計算すればよい, ということですから, □に当てはまる語は, 次のようなコードで計算することができます.

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

*84 研究レベルでは「右打ち切り可能」, すなわち左から重要な次元が並んでおり, どこで打ち切ってもそこまでで最適な性能が出るように単語ベクトルを学習する方法も提案されています[97]. また, 3.5.4 節で説明する行列分解による単語ベクトルは, 主成分分析を利用しているため, 同様に最も重要な固有ベクトルから順に用いた単語ベクトルを計算することができます.

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

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

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

こちらも, 学習するテキストを増やすとより正確になります [*85]. こうした単語ベクトルの演算はすべて, 単語ベクトルの長さを 1 に規格化した上で行っている (単位超球面上のベクトルとして計算を行っている) ことに注意してください.

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

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

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

- 単語ベクトルはどうやって学習されるのか

---

*85　松田聖子に対応するアメリカの歌手として 1 位になった “リンダ” は, 形態素解析の都合で切れていますが, 「リンダ・ロンシュタット」のことではないかと考えられます.

- 学習された単語ベクトルには, なぜこうした性質が成り立つのか

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

3.5.3 単語ベクトルの学習

(a) 連続的単語集合 (CBOW)

図 3.27 のニューラル n グラム言語モデルでは実験によって, 式(3.81)の隠れ層 $\mathbf{h}$ を使わず, 式(3.80)の $\mathbf{x}$ からの対数線形モデルだけでも十分な性能が出ることがわかっていました. そこで, 文脈語の単語ベクトルを連結して $\mathbf{x}$ とするかわりに, 図 3.32(a) に示したように同じ次元をもつベクトルに足し合わせたものを $\mathbf{x}$ とし, これを使って単語 w_t を予測できるようにすれば, より簡単に単語ベクトルを学習することができます. 文脈 $w_{t-C}, \cdots, w_{t-1}$ の順番を考えず, それらを $\mathbf{x}$ に足し込むため, これは単語の順番を考慮せず, 袋 (bag) づめにした集合とみる単語集合 (5 章) の連続版とみなすことができるため, これを**連続的単語集合** (continuous bag of words, **CBOW**) モデルといいます.

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

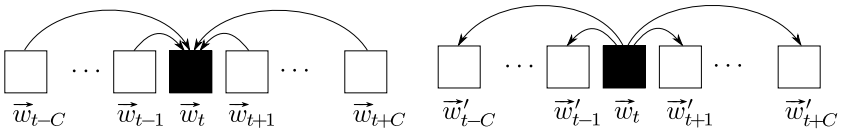
先月 自民党 は □

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

先月 自民党 は □ に 法案 を

とわかっていれば, □の単語はほぼ「国会」と決定することができるでしょう. よって実際には, CBOW は図 3.32(a) に示したように, 後続する単語 $w_{t+1}, w_{t+2}, \cdots, w_{t+C}$ も使って計算します. すなわち式で書けば, $n(t)$ を時刻 t の前後 C 個の文脈窓の添字として

$$\begin{cases} \mathbf{x}_t = \sum\limits_{i \in n(t)} \vec{w}_i \qquad (n(t) = \{ t-C, \cdots, t-1, t+1, \cdots, t+C \}) \\ \mathbf{p}_t = \mathrm{Softmax}\,(\mathbf{A}\mathbf{x}_t) \end{cases} \tag{3.93}$$



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

図 3.32: Word2Vec での単語ベクトルの学習方法. 単語ベクトル $\vec{w}_t$ は, (a) 周囲の単語ベクトルから自分を予測できるか, または (b) 自分から周囲の単語ベクトルを予測できるか, のどちらかを目的にして学習されます.

としたとき, 目的関数

$$\prod_{t=1}^{T} p_t(w_t) \tag{3.94}$$

が最大となるように $\vec{w}$ を学習します. [*86]

CBOW では文脈の単語の順番を考慮しないため, n グラムモデルと違って組み合わせ爆発 (次元の呪い) が発生しないので, 文脈の長さ C を大きくとることができます. 通常は C は 10 以下の値を用いますが, C を大きくとりすぎると「意味がぼやける」ため, 適切な値に設定する必要があります. また, 文脈として前後の単語ではなく, 係り受け関係にある (離れている可能性もある) 語を使うこともできます. その場合には単語ベクトルとして, より文法的な機能が重視されたベクトルが得られることになり, 構文解析などに有用であることが確かめられています[98]. このように, 使う「文脈」は, ベクトルの用途によって本来異なることに注意してください. 通常のスキップグラムのように前後の単語を使う場合でも, **C が小さいほど文法的な関係が, C が大きいほど意味的な関係が重視**された単語ベクトルが学習されることになります.

(b) スキップグラム

上の CBOW では周囲の単語ベクトルの平均から, 中心の単語 w_t を予測することを目的として単語ベクトルを学習していました. もう一つのアプローチとし

*86　なお, このことで n グラムと異なり, CBOW は単語列の生成モデルではなくなってしまいます. 数学的には, 周囲が与えられた場合の中心の語の確率が与えられている, イジングモデル[30, 31 章]と同様なマルコフ確率場を考えていることになります. 生成モデルからマルコフ確率場への定式化の変化は, 深層学習による GPT-3 のような現在の強力なマスク化言語モデルにも受け継がれています.

て, 図 3.32(b) に示したように, 逆に w_t から周囲の単語を予測する, という方法も考えられます. つまり,

$$\prod_{t=1}^{T} \prod_{j \in n(t)} p(w_j \,|\, w_t) \tag{3.95}$$

を最大化することが目的となります. このとき, 上の確率を単語ベクトル $\vec{w}_t$ と $\vec{c}_j$ の内積 $\vec{w_t} \cdot \vec{c_j}$ を使って,

$$\begin{cases} p(w_j \,|\, w_t) = \dfrac{\exp(\vec{w_t} \cdot \vec{c_j})}{Z} \\ Z = \displaystyle\sum_{j=1}^{V} \exp(\vec{w_t} \cdot \vec{c_j}) \end{cases} \tag{3.96}$$

とモデル化します. 式(3.96)では, w_t と周辺の語とのバイグラムを, 必ずしも隣りあっていない何語かスキップした場合も含めて考えるため, これを**スキップグラム** (skip-gram) といいます. なお, スキップグラムでは目的とする単語ベクトル $\vec{w}$ と, 周辺語のベクトル $\vec{c}$ は別のものを用います. すなわち, 各単語 j について $\vec{w_j}$ と $\vec{c_j}$ の二種類のベクトルを考えることになります.

スキップグラムでは, CBOW と異なり, 周辺語の単語ベクトルを $\mathbf{x}$ に平均化することなく, 式(3.96)のように別々に $\vec{w_t}$ と $\vec{c_j}$ の内積をとって単語ベクトルを最適化するため, より単語ベクトルの表現力が高まることが期待されます.

ただし, ナイーブに式(3.96)を用いると, 計算量が非常に大きくなってしまいます. これは, 式(3.96)の分母の正規化定数である Z[*87] が, $\vec{w_t}$, $\vec{c_j}$ が最適化で変わるごとにすべて計算し直さねばならず, この和は語彙に含まれる V 個の単語すべてにわたる和で, 通常 V は少なくても 1 万, 多いと 10 万からそれ以上にもなるからです.

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

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

Word2Vec の計算は **SGNS** (Skip-Gram with Negative Sampling) とよばれ, 単語ベクトルのもっとも標準的な学習法になっています.

負例サンプリング　式(3.96)のスキップグラムで単語ベクトルを効率的に学習する方法として, 機械学習で用いられているノイズ対照推定[99]の考え方を用いて,

- 実際に出現した語 c の確率を大きく, かつ
- それ以外の任意の語 c' の確率を小さく

することが考えられます. すなわち, 各 c ごとに複数の c' があると考えて

$$L = \prod_{c \in n(w)} \left[\sigma(\vec{w} \cdot \vec{c}) \times \prod_{c'} (1 - \sigma(\vec{w} \cdot \vec{c}\,')) \right] \tag{3.97}$$

を最大化することを考えます. 第 2 項は観測されなかった語 c' に対する積ですが, これを 1 つの観測値 c ごとに k 個あると考えて [*88], 単語分布 $p^*(c)$ からとることにすれば, 式(3.97)は

$$L = \prod_{c \in n(w)} \left[\sigma(\vec{w} \cdot \vec{c}) \times \prod_{i=1}^{k} \mathbb{E}_{c' \sim p^*(c)} \left[1 - \sigma(\vec{w} \cdot \vec{c}\,') \right] \right] \tag{3.98}$$

となります. この対数をとって [*89],

$$\log L = \sum_{c \in n(w)} \left[\log \sigma(\vec{w} \cdot \vec{c}) + k \cdot \mathbb{E}_{c' \sim p^*(c)} \left[\log(1 - \sigma(\vec{w} \cdot \vec{c}\,')) \right] \right] \tag{3.99}$$

を最大化することにしましょう. 1 回の最適化では 1 つの c あたり k 個の c' しか負例として考慮しませんが, 一般にこの最適化を繰り返すため, 学習の過程ではさまざまな単語 c' が負例として登場することに注意してください. 負例の数 k は, $\vec{w}$ を求めるための情報が少ない小さいコーパスの場合は 5 ～ 20 個程度, 大きなコーパスでは 2 ～ 5 個程度でよいことが原論文で報告されています.

負例をサンプリングする分布 $p^*(c)$ としては, 一様分布 $1/V$ やコーパスから

---

*88　すなわち, ここでは観測値とノイズが比率 $1:k$ の混合分布から生成されたと考えています[99].

*89　原論文では式(3.99)のように説明されていますが, 厳密には式(3.98)の対数をとると log は $\mathbb{E}$ の外にありますから, 式(3.99)は Jensen の不等式を使った式(3.98)の下界になっています.

計算するユニグラム分布 $p(c)$ など, さまざまな可能性があります. ただし, 一様分布では「奏覧」や「卯原内」といった極端に稀な単語も同等に出現してしまいますし [*90], いっぽうユニグラム分布を使うと, 3.2 節でみたように「が」「の」といった, ごく一部の機能語ばかりが確率が高いためにサンプリングされることになってしまいます. 実験によれば, この間をとって, ユニグラム分布を 3/4 乗して平滑化した

$$p^*(v) = \frac{p(v)^{3/4}}{Z} \qquad \left(Z = \sum_{v=1}^{V} p(v)^{3/4}\right) \tag{3.100}$$

を使うとよい結果になることが確かめられています.

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

$$r(v) = \max\left(1 - \sqrt{\frac{a}{p(v)}}, 0\right) \tag{3.101}$$

の確率で, 共起の計算前に文から単語を削除するヒューリスティックが使われています.[*91] ここで a は 10^{-5} 程度の閾値で, 単語の確率 $p(v)$ がこれより低い単語は削除されず, これより高いと上の確率で削除されることになります. この関数の形については, 4 章の図 4.4 を参照してください. たとえば, Brown コーパスでは $p(\texttt{the}) = 0.069$, $p(\texttt{hexagonal}) = 0.00000198$ なので, $a = 10^{-5}$ のとき

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

となり, 式(3.101)のルールでは “the” は 96%の確率で文から削除されること

*90 3.2 節の Zipf の法則によって, こうした頻度の低い語は非常に多く存在することに注意してください. 実際には, 語彙のほとんどはこうした語からなっているため, 一様分布からのサンプルはほとんどが稀な単語で占められてしまうことになります.

*91 原論文[76]では $p(v)$ は v の頻度となっており, その場合は式(3.101)は無意味な値となるため, 誤っていることに注意が必要です. 実際の `word2vec.c` や gensim の実装ではまた違った式が使われており, これに理論的な根拠があるわけではありません. より理論的な方法については, 4.2.2 節の SIF 単語重みの議論を参照してください.

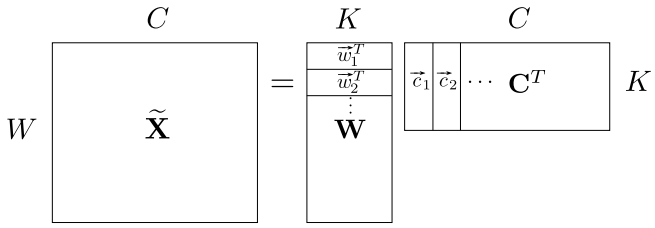


図 3.33: 行列の特異値分解 (SVD) によるニューラル単語ベクトルの学習．Word2Vec で学習される単語ベクトルは, Shifted PPMI (本文参照) を要素とする行列 $\widetilde{\mathbf{X}}$ を $\widetilde{\mathbf{X}} \simeq \mathbf{W}\mathbf{C}^T$ と特異値分解して得られるベクトルと, 数学的に等価です．

になり, 一方 “hexagonal” はまったく削除されないことになります．

Word2Vec では, テキストの単語をこうしてあらかじめ確率的に削除してから計算を行っています．これにより, “the” のような単語が削除されて共起窓が実質的に広がるため, より意味を考慮した共起データを得ることができ, 性能が改善することが確かめられています[100]．

3.5.4　Word2Vec と行列分解

前の節で計算したニューラル単語ベクトルは, 数学的には何を計算していることになるのでしょうか．イスラエルの Levy と Goldberg [101]は 2014 年に, Word2Vec で得られる単語ベクトルは, データから計算される**ある行列の主成分分析 (特異値分解) と数学的に等価である**ことを示しました．これにより, 実はニューラル単語ベクトルは, **行列の特異値分解で学習する**ことができます．ここからは行列の計算を用いますので, 線形代数に慣れていない方は, 本章末の文献案内で基礎を復習しておいてください．

Word2Vec のスキップグラムで最大化している目的関数は, 式(3.99) でした．この式を最適化することで, 単語ベクトル $\vec{w}$ と周辺語ベクトル $\vec{c}$ が計算できますが, 式(3.99)で, $\vec{w}$ と $\vec{c}$ はつねに $\vec{w} \cdot \vec{c} = \vec{w}^T \vec{c}$ の形でしか現れません．すべての $\vec{w}^T \vec{c}$ の組み合わせは, 図 3.33 に示したように $\vec{w_1}^T, \vec{w_2}^T, \cdots$ を縦に並べた行列を $\mathbf{W}$, また $\vec{c_1}, \vec{c_2}, \cdots$ を横に並べた行列を $\mathbf{C}^T$ とおいて

$$\mathbf{X} = \mathbf{W}\mathbf{C}^T \tag{3.102}$$

の要素になっています. よって, 式(3.99)は本質的にこの行列 $\mathbf{X}$ を最適化しており, それを式(3.102)のように行列分解すれば, 単語ベクトルの行列 $\mathbf{W}$ と周辺語ベクトルの行列 $\mathbf{C}$ が得られる, ということがわかります. それでは, この行列 $\mathbf{X}$ はどんな行列でしょうか.

$\mathbf{X}$ の (w,c) 要素は内積 $\vec{w}\cdot\vec{c}$ ですから, これを $x = X(w,c) = \vec{w}\cdot\vec{c}$ とおけば, ある x についての式(3.99)の目的関数 $L(x)$ は, 式(3.89)でみたように $1-\sigma(x)=\sigma(-x)$ ですから

$$\log L(x) = n(w,c)\log\sigma(x) + k\cdot n(w)\,p(c)\log\sigma(-x) \tag{3.103}$$

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

$$\sigma(x)' = \left(\frac{1}{1+e^{-x}}\right)' = -\frac{-e^{-x}}{(1+e^{-x})^2} = \sigma(x)\sigma(-x) \tag{3.104}$$

$$\sigma(-x)' = \left(\frac{1}{1+e^{x}}\right)' = -\frac{e^{x}}{(1+e^{x})^2} = -\sigma(x)\sigma(-x)$$

が成り立つことに注意しましょう. よって,

$$(\log\sigma(x))' = \frac{1}{\sigma(x)}\cdot\sigma(x)\sigma(-x) = \sigma(-x) \tag{3.105}$$

$$(\log\sigma(-x))' = \frac{1}{\sigma(-x)}\cdot -\sigma(x)\sigma(-x) = -\sigma(x)$$

です. この関係を使って式(3.103)を x で偏微分すれば, 最大になる点で x に関する勾配は 0 になりますから,

$$\frac{\partial}{\partial x}\log L(x) = n(w,c)\,\sigma(-x) + k\cdot n(w)\,p(c)\cdot(-\sigma(x)) = 0 \tag{3.106}$$

が成り立ちます. よって, ふたたび $\sigma(-x)=1-\sigma(x)$ を用いて,

$$n(w,c)(1-\sigma(x)) - k\cdot n(w)\,p(c)\sigma(x) = 0 \tag{3.107}$$

$$\therefore\quad \sigma(x) = \frac{n(w,c)}{n(w,c) + k\cdot n(w)\,p(c)} \quad (\equiv y)$$

となります. この右辺を y とおくと,

$$\sigma(x) = \frac{1}{1+e^{-x}} = y \quad \text{を解いて} \quad x = -\log\left(\frac{1}{y} - 1\right) \tag{3.108}$$

です. y をもとに戻して整理すれば,

$$\begin{aligned} x &= -\log\left(\frac{n(w,c) + k \cdot n(w)\, p(c)}{n(w,c)} - 1\right) \\ &= \log \frac{n(w,c)}{k \cdot n(w)\, p(c)} = \log \frac{\frac{n(w,c)}{N}}{\frac{n(w)}{N}\, p(c)} - \log k \\ &= \log \frac{p(w,c)}{p(w)\, p(c)} - \log k \end{aligned} \tag{3.109}$$

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

$$\mathrm{PMI}\,(w,c) = \log \frac{p(w,c)}{p(w)\, p(c)} \tag{3.110}$$

を, $\log k$ だけシフトしたものになっています. したがって, Word2Vec で学習している式(3.102)の行列 $\mathbf{X}$ は,

$$\begin{aligned} X(w,c) &= \mathrm{PMI}\,(w,c) - \log k \\ &= \log \frac{p(w,c)}{p(w)\, p(c)} - \log k \end{aligned} \tag{3.111}$$

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

ただし, 式(3.111)の行列は, ほとんどを占める $p(w,c) = 0$ の場合に値が $\log p(w,c) = -\infty$ になってしまいます. 要素がすべて埋まると計算量も増えてしまうため, 実際に現れる (w,c) のペアのほとんどは $\mathrm{PMI}\,(w,c) > 0$ である (だからこそ出現した) ことから, 式(3.111)で値が非負の場合だけを考えて [*92],

*92　PMI が負の場合を考えないことで, 正確には目的関数は式(3.103)とは異なってしまいますが, 実験的には以下で示すように, これは Word2Vec のかなり良い近似になっていることがわかっています.

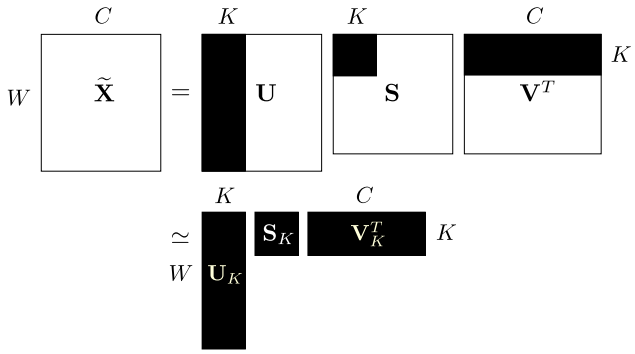


図 3.34: SPPMI 行列 $\widetilde{\mathbf{X}}$ の特異値分解 (SVD) による次元削減の様子.

$$\widetilde{X}(w,c) = \max\left(\log\frac{p(w,c)}{p(w)\,p(c)} - \log k, 0\right) \tag{3.112}$$

を用いることにします. これを PPMI (Positive PMI), または SPPMI (Shifted PPMI) とよびます [*93].

SPPMI を並べた行列 $\widetilde{\mathbf{X}}$ は, カウントが 0 のエントリ (w,c) および式(3.111)が負になるエントリはすべて値が 0 になるため, 非常に疎になることに注意してください. 実際, `ja.text8.txt` で窓幅 10 の場合, 非 0 のエントリは 1% (20,872,503/2,027,160,576) にすぎず, 99%のエントリが 0 になりました. $\widetilde{\mathbf{X}}$ から式(3.102)の $\mathbf{W}, \mathbf{C}$ を求めるには, こうした疎行列に対する**特異値分解** (Singular Value Decomposition, **SVD**) を使うと高速に計算することができます.

SVD は行列の対角化を正方行列でない場合に拡張したもので, 行列 $\widetilde{\mathbf{X}}$ を図 3.34 の上段のように

$$\widetilde{\mathbf{X}} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{3.113}$$

と 3 つの行列 $\mathbf{U}, \mathbf{S}, \mathbf{V}^T$ の積に分解します. ここで $\mathbf{U}, \mathbf{V}$ はそれぞれ $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T$ お

*93 PPMI はこの研究で初めて現れたわけではなく, 5 章で扱う潜在意味解析 (LSA) の文脈で, 2007 年に Bullinaria らによって提案され[102], 他のさまざまな単語ベクトルに比べて, PPMI による単語ベクトルとコサイン距離による比較が最も高い性能を持つことが報告されていました. 2012 年には SVD による次元圧縮も試みられており[103], Word2Vec と本質的に同じベクトルが, 深層学習より以前にすでに提案されていたといえます.

よび $\widetilde{\mathbf{X}}^T\widetilde{\mathbf{X}}$ の固有ベクトル, $\mathbf{S}$ は対応する固有値の平方根 (特異値) $\sigma_1, \sigma_2, \cdots, \sigma_r$ (r は $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T, \widetilde{\mathbf{X}}^T\widetilde{\mathbf{X}}$ のランクです) を並べた対角行列

$$\mathbf{S} = \begin{pmatrix} \sigma_1 & & & \mathbf{O} \\ & \sigma_2 & & \\ & & \ddots & \\ \mathbf{O} & & & \sigma_r \end{pmatrix} \tag{3.114}$$

です. このうち特異値の大きい方から上位 K 個をとって, 図 3.34 の下段のように

$$\widetilde{\mathbf{X}} \simeq \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^T \tag{3.115}$$

とする近似は, $\widetilde{\mathbf{X}}$ との二乗誤差を最小にする近似になっています[104, §3.4]. 式(3.115)は, $\mathbf{S}$ は対称行列なので $\mathbf{S}^T = \mathbf{S}$ で, 一般に $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ ですから,

$$\begin{aligned} \widetilde{\mathbf{X}} &\simeq \mathbf{U}_K \mathbf{S}_K^{1/2}\, \mathbf{S}_K^{1/2} \mathbf{V}_K^T \\ &= \left(\mathbf{U}_K \mathbf{S}_K^{1/2}\right) \left(\mathbf{V}_K \mathbf{S}_K^{1/2}\right)^T \quad (= \mathbf{W}\mathbf{C}^T) \end{aligned} \tag{3.116}$$

とも書くことができます. よって, 式(3.102)と見比べて

$$\mathbf{W} = \mathbf{U}_K \mathbf{S}_K^{1/2}, \quad \mathbf{C} = \mathbf{V}_K \mathbf{S}_K^{1/2} \tag{3.117}$$

とおけば, $\mathbf{W}, \mathbf{C}$ を得ることができます [*94]. ここで $\mathbf{S}_K^{1/2}$ は, $\mathbf{S}$ の対角成分の平方根をとった行列

$$\mathbf{S}^{1/2} = \begin{pmatrix} \sqrt{\sigma_1} & & & \mathbf{O} \\ & \sqrt{\sigma_2} & & \\ & & \ddots & \\ \mathbf{O} & & & \sqrt{\sigma_r} \end{pmatrix} \tag{3.118}$$

---

*94　行列 $\widetilde{\mathbf{X}}$ の各行, すなわち各単語の共起情報を格納したベクトルの間の内積は $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T$ で計算することができます. $\widetilde{\mathbf{X}} \simeq \mathbf{U}_K\mathbf{S}_K\mathbf{V}_K^T$ ですから, これは $\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T = \mathbf{U}_K\mathbf{S}_K\mathbf{V}_K^T(\mathbf{U}_K\mathbf{S}_K\mathbf{V}_K^T)^T = \mathbf{U}_K\mathbf{S}_K\mathbf{V}_K^T\mathbf{V}_K\mathbf{S}_K^T\mathbf{U}_K^T = (\mathbf{U}_K\mathbf{S}_K)(\mathbf{V}_K\mathbf{S}_K)^T$ となり, $\mathbf{W} = \mathbf{U}_K\mathbf{S}_K, \mathbf{C} = \mathbf{V}_K\mathbf{S}_K$ とするのが $\widetilde{\mathbf{X}}$ の各行間の内積を保存する意味では理論的に最適です. しかし, 式(3.116)のように直接行列分解を近似する方が, さまざまな意味的タスクにおいて精度が高いことが確かめられています[100].

です. Python では,

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

のように実行すれば, $\mathbf{W}, \mathbf{C}$ を計算することができます [*95].

単語ベクトルの計算 式(3.110)の PMI は, 式(3.109)から, 頻度 $n(w,c)$, $n(w)$, $n(c)$ を用いて

$$\begin{aligned}(3.119)\qquad \mathrm{PMI}\,(w,c) &= \log \frac{p(w,c)}{p(w)\,p(c)} = \log \frac{n(w,c)/N}{n(w)/N \cdot n(c)/N} \\ &= \log n(w,c) - \log n(w) - \log n(c) + \log N\end{aligned}$$

で計算することができます. 式(3.112)から, この値が $\log k$ より大きくないと値が 0 になり, 文脈語 c が観測されなかったことになってしまいますので, [100] では実験の結果, $k=1$ すなわち $\log k = 0$ として PPMI をそのまま使う方が意味的タスクにおいて高性能となることが確かめられています [*96]. また式(3.110)は, 条件つき確率の定義から

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

とも書くことができることに注意してください. すなわち $\mathrm{PMI}\,(w,c)$ は, 「w の周辺に c が現れる確率」と「c が一般的に出現する確率」の比の対数になっています [*97]. このとき, c が稀な単語で $p(c)$ が非常に小さいと, PMI の値が非常に大きくなってしまいます. これを避けるため, $p(c)$ としては負例サンプリングでも用いた, 式(3.100)で平滑化した $p^*(c)$ を用いるとよいでしょう[100]. あらかじめ式(3.100)の対数 $\log p^*(c)$ を計算しておけば, 平滑化された式(3.120)は

$$(3.121)\quad \log \frac{p(c|w)}{p^*(c)} = \log \frac{n(w,c)/n(w)}{p^*(c)} = \log n(w,c) - \log n(w) - \log p^*(c)$$

*95 SciPy の `svds` は, `S` に含まれる特異値を降順ではなく昇順で返す (公式には, 順番は保証されていない) ので, 実装の際には注意してください.

*96 Word2Vec を使う場合との違いは, 最適化の方法によるものと考えられます.

*97 これは, **対数尤度比**ともよばれています.

```
の             -0.00170  0.00917  0.00827  0.00624 -0.01122 -0.00228  0.00026 ..
、              0.00459 -0.00642 -0.01133  0.01042 -0.00060 -0.00573  0.00493 ..
:
能登線          0.04051 -0.00000 -0.04853  0.05437 -0.07357  0.00426 -0.00886 ..
クリーマー      0.03077  0.03630 -0.02890  0.02567 -0.02269  0.05661  0.04827 ..
鞠子           -0.02448  0.05607 -0.00669 -0.00328  0.01572 -0.01031  0.02368 ..
奈良教育大学    0.09819 -0.03328 -0.05696  0.01821 -0.06707 -0.02212  0.06239 ..
エスプラナーデ -0.04707 -0.03423  0.02346 -0.05047 -0.01056  0.09426 -0.00005 ..
```

図 3.35: 単語間共起行列の行列分解から `pmivector.py` で計算された単語ベクトルの例.

で求めることができます.

こうして求めた PMI 行列から計算した単語ベクトルの例を, 図 3.35 に示しました. この計算は, サポートページの `pmivector.py` を使って,

```
% pmivector.py -K 100 -w 10 ja.text8.txt wordvector.vec
counting lexicon..
counting cooccurrences..
processing lines 516000..
datalen = 14729961, lexicon = 45024, k = 1
creating sparse matrix 45024/45024..
creating COO matrix..
computing SVD..
done.
```

のようにすると行うことができます [*98]. Word2Vec の計算に比べると若干時間がかかりますが, 数学的な最適解が求められること, および Word2Vec とは異なり, SVD によって単語ベクトルの次元が重要な順に並ぶことが特徴といえます. なお, 共起行列 $\widetilde{\mathbf{X}}$ が巨大になる場合は, ランダム射影を用いて SVD を効率的に行うことのできる redsvd[*99] のような方法を使えば, データが大きくても高速に単語ベクトルを計算することができます.

自己相互情報量と言語モデル　Word2Vec の標準的な学習方法である SGNS は $\mathrm{PMI}(w, c)$ を近似していることがわかりましたが, 言語の生成モデルとしては, これは何を意味しているのでしょうか. 式(3.120)でみたように

$$\mathrm{PMI}(w, c) = \log \frac{p(w, c)}{p(w)\, p(c)} = \log \frac{p(c|w)}{p(c)} \tag{3.122}$$

*98　この計算には, 執筆時の環境では 6 分程度かかりました.
*99　`https://code.google.com/archive/p/redsvd/`

ですから, 両辺を指数の肩にのせれば,

$$\exp(\mathrm{PMI}\,(w,c)) = \frac{p(c|w)}{p(c)} \tag{3.123}$$

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

となります. 式(3.123)は, 単語 w の周りに文脈語 c が現れる条件つき確率 $p(c|w)$ は, c のデフォルトの出現確率 $p(c)$ に, 係数 $e^{\mathrm{PMI}\,(w,c)}$ をかけたものになることを表しています. $\mathrm{PMI}\,(w,c)=0$, すなわち式(3.110)より $p(w,c)=p(w)p(c)$ で w と c が無相関の場合は, c の条件つき確率は $p(c)\cdot e^0=p(c)\cdot 1=p(c)$ でデフォルトの確率に等しく, $\mathrm{PMI}\,(w,c)>0$ なら確率は $e^{\mathrm{PMI}\,(w,c)}>1$ なので $p(c)$ より大きく, $\mathrm{PMI}\,(w,c)<0$ なら確率は $e^{\mathrm{PMI}\,(w,c)}<1$ なので $p(c)$ より小さくなるわけです. このように, 自己相互情報量は**確率を式(3.123)のように「変調」する係数** (の対数) を表しており, これは統計学では Cox 比例ハザード[105]とよばれるモデルと同じモデルだといえます.

3.5.5* GloVe と意味方向の数理

こうして得られた単語ベクトルの間には, 3.5.1 節で示した「引き算」の関係が成り立つことが知られていますが, これはなぜなのでしょうか. スタンフォード大学で素粒子物理を専攻していた Pennington は 2014 年に, こうした意味的関係が成り立つように設計された単語ベクトル, GloVe (Global Vector) [106] を提案しました. GloVe の単語ベクトルは Word2Vec の単語ベクトルと似ていますが, 目的関数が少し異なり, 若干違うベクトルになっています.

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

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

$$p(c|w) = \frac{n(w,c)}{n(w)} \tag{3.124}$$

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

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

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

$$\frac{p(k|i)}{p(k|j)} \tag{3.125}$$

をさまざまな k について計算すると, 表3.6 のようになります. これから, 図 3.36 に示したように

- 犬に強く関係する単語 k, たとえば “飼育” は $p(k|犬)$ の方が $p(k|猫)$ よりも大きく,
$$\frac{p(飼育|犬)}{p(飼育|猫)} = 5.803 \gg 1$$
のように, 比が 1 よりもずっと大きくなります.
- 逆に, 猫に強く関係する単語 k, たとえば “彼女” は $p(k|猫)$ の方が $p(k|犬)$ より大きく,
$$\frac{p(彼女|犬)}{p(彼女|猫)} = 0.186 \ll 1$$
のように, 比が 1 よりずっと小さくなっています.
- 一方, 犬とも猫とも関係のある単語 “動物”, およびどちらとも関係ない単語 “本陣” は,
$$\frac{p(動物|犬)}{p(動物|猫)} = 1.444 \approx 1$$
$$\frac{p(本陣|犬)}{p(本陣|猫)} = 0.929 \approx 1$$
と, どちらも 1 に近くなっています (ただし, 表3.6 にみるように分子・分母の確率の絶対値は関係があれば大きく, なければ小さくなります).

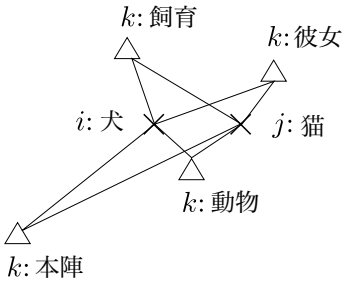


図 3.36: GloVe の用いる単語ベクトルの間の関係. それぞれの単語 k が単語 i, j の周辺に現れる確率 $p(k|i)$, $p(k|j)$ の比 $p(k|i)/p(k|j)$ から, 単語ベクトルを求めます.

つまり一般に, 図 3.36 のような単語の間の意味的関係は

$$\frac{p(k|i)}{p(k|j)} \equiv \frac{p_{ik}}{p_{jk}} \tag{3.126}$$

の値で示される, ということがわかります. ここで簡単のため, $p(k|i)=p_{ik}$, $p(k|j)=p_{jk}$ と表記しました. 式(3.126)は確率の比, すなわち相対値を見ているため, 個々の確率の絶対値 p_{ik}, p_{jk} が単語の頻度によって変わっても不変な値であることに注意してください. したがって, これから求める単語ベクトル $\overrightarrow{w_i}, \overrightarrow{w_j}, \widetilde{w_k}$ から式(3.126)が計算されるべきですから, その関数を一般に F とおけば,

$$F(\overrightarrow{w_i}, \overrightarrow{w_j}, \widetilde{w_k}) = \frac{p_{ik}}{p_{jk}} \tag{3.127}$$

が計算できて図 3.36 の関係が成り立つべきだ, ということになります. ここで, GloVe では Word2Vec の文脈語ベクトル $\vec{c}$ と同様に, 共起する語 k については別の種類のベクトル $\widetilde{w_k}$ を推定します.

関数 F にはいろいろな可能性がありますが, 3.5.2 節で示した「ベクトルの引き算」が成り立つためには, 最も単純には F は差 $\overrightarrow{w_i} - \overrightarrow{w_j}$ に依存する関数となるべきでしょう. また, 式(3.127)の右辺はスカラー値ですから, 線形な構造を保存してベクトルの次元の間に無用な相関を持ち込まないためには, $\overrightarrow{w_i} - \overrightarrow{w_j}$ と $\widetilde{w_k}$ の内積をとればよいと考えられます. よって, 式(3.127)は

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

と書き換えることができました.

ここで, 式(3.128)の右辺は確率の比のため, 必ず正であることに注意してください. よって F は必ず正の値を返す必要がありますが, 左辺の ( ) の中は $\overrightarrow{w_i}^T\widetilde{w_k} - \overrightarrow{w_j}^T\widetilde{w_k}$ と差になっており, 負になる可能性がありますから, 一般に実数の和や差を正の数の積や割り算に変換できる F を考えれば [*100], 式(3.128)はさらに

$$F\left((\overrightarrow{w_i} - \overrightarrow{w_j})^T\widetilde{w_k}\right) = \frac{F\left(\overrightarrow{w_i}^T\widetilde{w_k}\right)}{F\left(\overrightarrow{w_j}^T\widetilde{w_k}\right)} = \frac{p_{ik}}{p_{jk}} \tag{3.129}$$

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

$$\exp\left(\overrightarrow{w_i}^T\widetilde{w_k}\right) = p_{ik} \tag{3.130}$$

すなわち

$$\overrightarrow{w_i}^T\widetilde{w_k} = \log p_{ik} = \log p(k\,|\,i) \tag{3.131}$$

が成り立てばよい, ということがわかります. 頻度を使って書き換えれば,

$$\overrightarrow{w_i}^T\widetilde{w_k} = \log n(i,k) - \log n(i) \tag{3.132}$$

となります. 式(3.132)が, これまでに述べた意味的な関係から単語ベクトルが満たすべき十分条件ということになります.

ここで, 式(3.132)の左辺は i と k を入れ替えても ($\overrightarrow{w}$ と $\widetilde{w}$ の違いを除けば) 成り立ちますが, 右辺は条件つき確率なので, i と k を逆にすると意味が変わってしまうことに注意してください. そこで $\log n()$ に対応するバイアス項 b_i および, 対称性から同様に $\widetilde{b}_k$ を導入すれば,

$$\begin{aligned} \overrightarrow{w_i}^T\widetilde{w_k} &= \log n(i,k) - b_i - \widetilde{b}_k \\ \therefore\quad \overrightarrow{w_i}^T\widetilde{w_k} + b_i + \widetilde{b}_k &= \log n(i,k) \end{aligned} \tag{3.133}$$

となり, i と k について対称な目的関数を得ることができました [*101].

---

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

*101　GloVe の真の目的関数は式(3.132)ですが, 最適化のために導入しているこうしたヒュー

ただし, 単純に式(3.133)の回帰モデルを求めるには問題があり, $n(i,k)=0$ の場合や, $n(i,k)>0$ でも, 偶然共起した場合をすべて説明しなければならなくなってしまいます. そこで, 式(3.133)の両辺の二乗誤差を, $n(i,k)$ に依存する関数 $f(n(i,k))$ で重みづけて, GloVe では

$$J=\sum_{i,k=1}^{V} f(n(i,k))\Big(\overrightarrow{w_i}^T\widetilde{w_k}+b_i+\widetilde{b}_k-\log n(i,k)\Big)^2 \tag{3.134}$$

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

$$f(x)=\begin{cases}\left(\dfrac{x}{x_{\max}}\right)^{\alpha} & x<x_{\max}\\ 1 & \text{それ以外}\end{cases} \tag{3.135}$$

という, 図 3.37 のような関数を用いると性能が良かったことが報告されています. なお, 本来は $\overrightarrow{w_i}$ と $\widetilde{w_k}$ の役割は同じですので, 最終的な単語ベクトルとしては平均をとった $(\overrightarrow{w_i}+\widetilde{w_k})/2$ が採用されています.

GloVe を使っても, Word2Vec とほとんど同様の単語類似度や比例関係を計算することができます (→演習 3–9). GloVe では, スタンフォード大学の公式ページで, Wikipedia 全体のような巨大なテキストから計算した英語の単語ベ

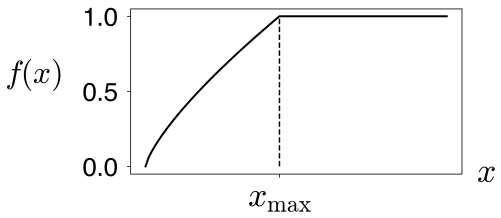
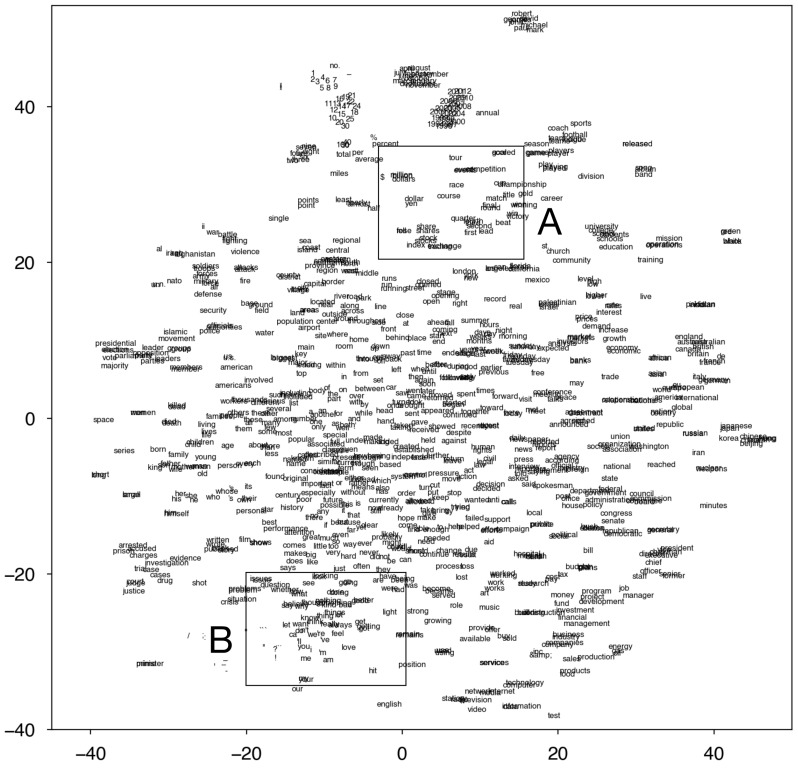
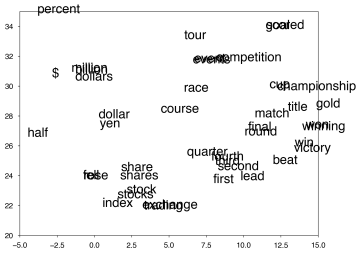


図 3.37: GloVe で用いている頻度 x の重みづけ関数 $f(x)$. $x=0$ では重み 0 に, $x_{\max}$ 以上ではすべて重みが 1 になります.

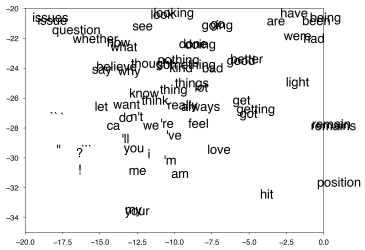
リスティックが, GloVe の性能を本来のものより下げている可能性があります. 直接, 式(3.132)を MCMC 法などで最適化することを考えてみてもよいでしょう.



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



(b) A を拡大したもの



(c) B を拡大したもの

図 3.38: GloVe で 6 億語のテキストから学習された頻度上位 1000 語の単語ベクトルの可視化. 100 次元空間の単語ベクトルを, t-SNE で 2 次元に可視化しています.

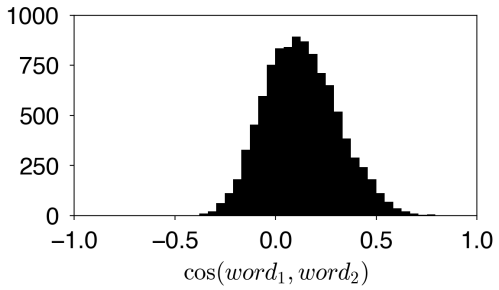


図 3.39: ja.text8 コーパスから計算した単語ベクトルでの, ランダムな 2 単語間のコサイン類似度の分布. 類似度は $[-1, 1]$ ではなく, その一部にだけ分布しています.

クトルが公開されています [*102]. このうち, 100 次元の単語ベクトルの頻度上位 1000 語を, t-SNE [13] とよばれる非線形な可視化法で 2 次元に可視化したものを図 3.38 に示しました [*103].

3.5.6* 単語ベクトルの分布とノルム

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

単語ベクトル $\vec{w}$ と $\vec{v}$ の類似度は, 3.5.2 節の「単語ベクトルの計算」でみたように, そのなす角のコサインとして

$$\cos(\vec{w}, \vec{v}) = \frac{\vec{w} \cdot \vec{v}}{|\vec{w}||\vec{v}|} \tag{3.136}$$

で測ることができます. 任意の実数 x について $-1 \leq \cos x \leq 1$ ですから, 式(3.92)は -1 から 1 の範囲に分布しているはずです. しかし, 次のようにランダムな 2 単語についてコサイン類似度を計算してみると, 図 3.39 のように最大値も最小値もまったく ± 1 ではなく, 平均値も 0 ではないことがわかります.

---

*102 https://nlp.stanford.edu/projects/glove/からダウンロードすることができます.
*103 この単語ベクトルの可視化は, サポートサイトの visualize.py を使うと行うことができます. 詳しくは, スクリプトの中身をご覧ください.

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

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

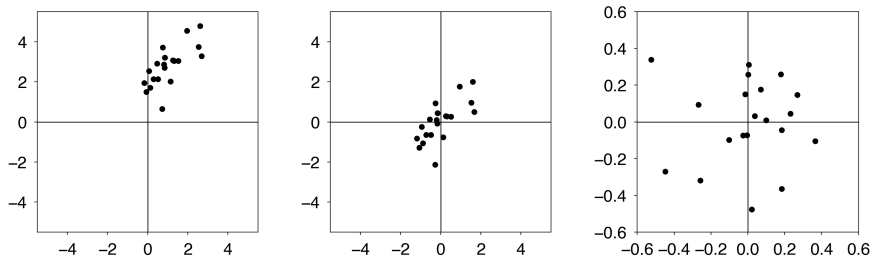
```
def loadvec (file):
    matrix = []
    with open (file, 'r') as fh:
        for line in fh:
            tokens = line.rstrip('\n').split()
            if len(tokens) > 2: # Word2Vec のヘッダをスキップ
                matrix.append (np.array (list ( \
                                map (float, tokens[1:]))))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

単語ベクトルの中心化 よって, 最も自然に考えられる前処理は, 単語ベクトルから上の中心を引いて, 平均を 0 にすることでしょう (図 3.40(b)).

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

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

ただし実は, 式(3.137)のようなナイーブな中心化をしてしまうと, 単語ベクトルの表現力はむしろ悪くなってしまうことが知られています. 3.2 節で議論し



(a) 元のデータ (b) 中心化を行ったもの (c) 白色化を行ったもの

図 3.40: ベクトルの中心化と白色化. 中心化により平均が 0 に, さらに白色化により共分散が単位行列 $\mathbf{I}$ になります.

たように, 言語には大量の単語があり, そのほとんどはきわめて低確率なのでした. 単純な平均をとると, 「日本」「人権」のような頻度の高い重要な単語ベクトルと, 「ザンペリーニ」や「鼠小僧次郎吉」のようなめったに出てこない特殊な単語ベクトルの平均をとることになってしまいます.

この問題に対し, 東北大学の横井ら[107, 108]は, 一様ではなく単語の確率で期待値をとって平均を計算することで, 単語ベクトルの性能が改善することを示しました. すなわち式(3.137)の代わりに,

$$\vec{w}' = \vec{w} - \mu \quad ; \mu = \sum_{v=1}^{V} p(v)\vec{v} \tag{3.138}$$

として単語ベクトルを中心化します. $p(v)$ は, 単語 v の出現確率です. サポートサイトに, この前処理を行うスクリプト `vcenter.py` を示しました. この処理には単語のユニグラム確率が必要ですので, 実行には

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

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

```
import re
```

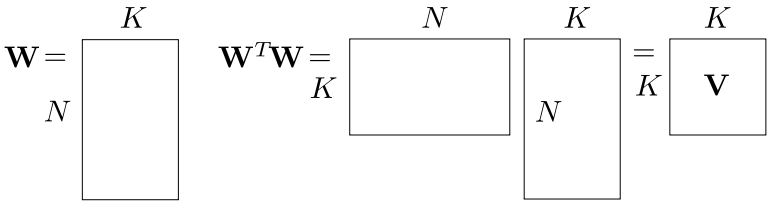


図 3.41: 単語ベクトルの行列 $\mathbf{W}$ とその共分散行列 $\mathbf{V}=\mathbf{W}^T\mathbf{W}$ の計算.

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

のようなジェネレータを定義すれば,

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

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

単語ベクトルの白色化 単語ベクトルの平均は中心化によって (期待値の意味で) 0 になりますが, 単語ベクトルの空間は図 3.40(b) のように, それでもまだ歪みを残しているはずです. こうした場合, データ解析でよく行われる前処理が**白色化** (whitening) です. 白色化は図 3.40(b) を図 3.40(c) のように線形射影によって変換する処理で, 平均を 0 にした上で, さらに異なる次元間の共分散を 0

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

ベクトルの白色化は, 一般に次のようにして行うことができます. いま, 各データが K 次元の行ベクトルで, N 個のデータが図 3.41 左のように, $N\times K$ の行列 $\mathbf{W}$ をなしているとします [*104]. $\mathbf{W}$ を中心化して行方向の平均を $\mathbb{E}[\mathbf{W}]=\mathbf{0}$ にしたとき, K 次元の各次元間の共分散行列は, 図 3.41 右のように $\mathbf{V}=\mathbf{W}^T\mathbf{W}$ で計算することができます. $(\mathbf{W}^T\mathbf{W})^T=\mathbf{W}^T\mathbf{W}$ より $\mathbf{V}$ は対称行列ですから, $\mathbf{V}$ は

$$\mathbf{V}=\mathbf{P}\mathbf{S}\mathbf{P}^{-1} \tag{3.139}$$

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

$$\mathbf{Z}=\mathbf{W}\mathbf{P}\mathbf{S}^{-1/2} \tag{3.140}$$

と $\mathbf{W}$ を $\mathbf{Z}$ に変換すれば [*105], その共分散は式(3.139)から,

$$\begin{aligned}\mathbf{Z}^T\mathbf{Z}&=(\mathbf{W}\mathbf{P}\mathbf{S}^{-1/2})^T\mathbf{W}\mathbf{P}\mathbf{S}^{-1/2}\\&=\mathbf{S}^{-1/2}\mathbf{P}^T\mathbf{W}^T\mathbf{W}\mathbf{P}\mathbf{S}^{-1/2}=\mathbf{S}^{-1/2}\underbrace{\mathbf{P}^T(\mathbf{P}}_{\mathbf{I}}\mathbf{S}\underbrace{\mathbf{P}^{-1})\mathbf{P}}_{\mathbf{I}}\mathbf{S}^{-1/2}\\&=\mathbf{S}^{-1/2}\mathbf{S}\mathbf{S}^{-1/2}=\mathbf{I}\end{aligned} \tag{3.141}$$

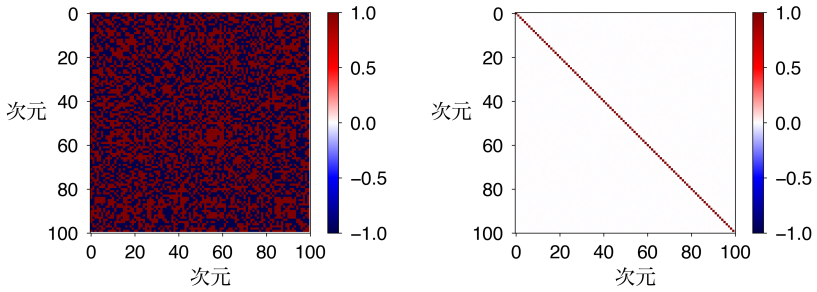
になります. すなわち, $\mathbf{W}$ を式(3.140)で線形変換した $\mathbf{Z}$ は平均 $\mathbb{E}[\mathbf{Z}]=\mathbf{0}$, 共分散 $\mathbb{V}[\mathbf{Z}]=\mathbf{I}$ で分布するようになります. これが**白色化**です.

したがって, 単語ベクトルを並べた行列 $\mathbf{W}$ についてこの白色化を行えば, 図 3.40(c) のようにデータは K 次元空間で無駄なく分布するようになり, 細かい意味の差を反映するようになると考えられます. ただし, ここでも行列 $\mathbf{W}$ の各行に対応する単語には大きな頻度の差があるため, [107]では式(3.138)の期待値を用いた中心化を行い, $\mathbf{W}$ の各行 $\overrightarrow{w_i}$ を, 長さを 1 に規格化した上で $\sqrt{p(w_i)}$ で重みづけた

---

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

*105 $\mathbf{S}^{-1/2}$ は, $1/\sqrt{\lambda_1},\cdots,1/\sqrt{\lambda_K}$ を対角成分にもつ対角行列です.



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

図 3.42: 単語ベクトルの共分散行列と白色化. 白色化により, 単語ベクトルの各次元を無相関にすることができます.

$$\overrightarrow{w_i}' = \sqrt{p(w_i)}\frac{\overrightarrow{w_i}}{|\overrightarrow{w_i}|} \tag{3.142}$$

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

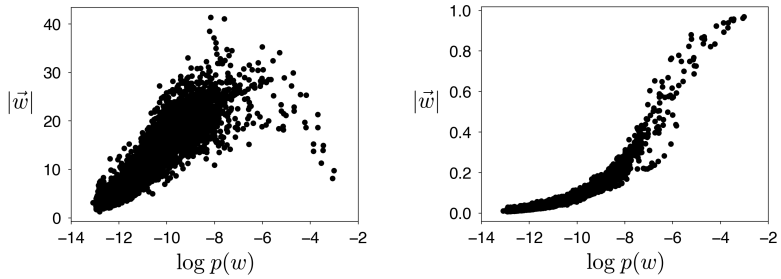
サポートサイトに, この方向白色化を行うスクリプト `vwhiten.py` を示しました.

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

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

図 3.42 に, もとの単語ベクトルの共分散行列 $\mathbf{V}$ と, 方向白色化後の共分散行列 $\mathbf{V}$ を示しました. もとの単語ベクトルでは各次元の間に $\pm$ の多くの相関がありますが, 変換後はそれが一掃され, 単語ベクトルの各次元はすべて無相関になっている (共分散が単位行列になっている) ことがわかります.

この新しい単語ベクトルで単語の類似度を計算してみると, 下のようになります. 3.5.2 節の「単語ベクトルの計算」での結果と比べてみると, “太陽” の類似語の上位から “シリウス” や “星雲” が外れていたり, “少年” の類似語の順番が入れ替わっていたりなど, 微妙にではありますが, 確実に類似度が改善していることがわかります. 数値的にも, 単語類似度や特に文類似度などのタスクの性



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

図 3.43: 単語の確率 (横軸) と単語ベクトルの長さ (縦軸) の関係.

能で改善がみられることが報告されています[107].

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

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

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

単語ベクトルの長さ $|\vec{w}|$ を縦軸に, その単語の出現確率 $p(w)$ の対数を横軸にとってプロットしてみると, 図 3.43 のように, 確率が中程度の単語ベクトルの長さが最も大きいことが知られています[109]. Word2Vec では確率的勾配法で単語ベクトルを最適化するため, 頻度の高い語のベクトルは様々な方向から「引っ張られる」ので, 結果的に長さは短くなるでしょう. また頻度の低い語は逆にわずかしか引っ張られませんので, 単語ベクトルは短いままになると考え

られます.

ただし, こうした頻度の影響を差し引いた上では, 意味が「強い」単語のベクトルはノルムが大きい傾向にあることが報告されています[110, 111]. ここで単語 w の意味が「強い」とは, w の周囲に出現する単語 c の確率分布 $\{p(c|w)\}$ と, 単語 c の平均的な確率分布 $\{p(c)\}$ との KL ダイバージェンス (式(2.70)) が大きい, すなわち周辺語の確率分布が偏っていることを意味します.

なお, 単語ベクトルの方向白色化を行うと, 頻度と長さの関係は図 3.43(b) のように単純な比例関係に変わります. Word2Vec だけでなく, GloVe や PMI の場合にどうなるかなど, 単語ベクトルの長さについては統一的な理論が待たれています (→演習 3–12).

3 章のまとめ

3 章では, 単語が従う Zipf の法則, Heaps の法則といった統計的な法則に続けて, 単語の n グラム言語モデルについて学習しました. 一般に有限個しかない文字に比べて, 単語の種類は原理的に無限にあるため, 言語的に正しくても観測頻度が 0 になるゼロ頻度問題が深刻になります. このための方法として, ディリクレ平滑化や Kneser-Ney 平滑化を説明しました. ディリクレ平滑化は単語の多項分布 $\mathbf{p}$ がディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成されたと考えると, 理論的に導くことができます. さらに $\mathbf{p}$ を積分消去したポリア分布を考えることで, $\boldsymbol{\alpha}$ を最適化できることも学びました. 現在は言語モデルとしては意味を考慮した深層学習が使われることが多くなっていますが, こうした方法は, 4 章の文モデルおよび 5 章の文書モデルの基礎になっています.

後半では, 最初の深層学習であるニューラル言語モデルから 2000 年代に生まれた単語ベクトルと, その理論的な背景について説明しました. 有名な Word2Vec は, 理論的には自己相互情報量 (PMI) の行列分解を行っているとみなすことができ, 実際に行列の特異値分解で最適解を計算することが可能です. PMI は単語のフレーズ化のような他の分析にも大変有効な統計量で, 5 章でもふたたび登場します. また, 意味の比例関係から数学的に導かれる単語ベクトル GloVe は, Word2Vec と似たベクトルになる結果もさることながら, その導出の仕方がたい

へん興味深いものです. 現在の自然言語の深層学習は, すべてこうした単語ベクトルを基盤としています.

3 章の演習問題

[3–1] 身近なテキストで, テキストの長さと語彙の大きさを図 3.4 のようにプロットしてみましょう. テキストのジャンルによって, どんな違いがあるでしょうか.

[3–2] 実際のテキストを使って, Heaps の法則の γ や Zipf の法則の α を線形回帰で求めてみましょう. 式(3.2)のように, 先に両辺の対数をとるのがポイントです. 線形回帰の係数の求め方については, [12]の 1 章などを参照してください.

[3–3] 単語と異なり, 文字は言語では一般に有限です. 有限集合である文字の場合にテキストを読みながら文字の「語彙」の大きさを計算した場合, Heaps の法則は成り立つでしょうか. 文字の種類の少ない英語と多い日本語では, 何か違いがあるでしょうか.

[3–4] テキストを単語に分けたとき, 頻度の低い単語はさまざまなノイズからなっていると考えられます. これらにはどのようなパターンがあるか, 頻度が 1 や 2 といった低い単語について調べてみましょう. 単語を正規化 (94 ページ) しても, 除けないものはあるでしょうか.

[3–5] 3.3 節の統計的フレーズ化を (大量の) プログラム言語のソースファイルに適用すると, どんなフレーズが得られるでしょうか. C 言語の `int main (int argc, char *argv[])` のような決まり文句は, 1 つの単語として結合されるでしょうか.

[3–6] 部分積分により, $\Gamma(x+1) = x\,\Gamma(x)$ が成り立つことを示してみましょう.

[3–7] カテゴリ数 K が大きい場合, ディリクレ分布 $\mathrm{Dir}(\mathbf{p}|\boldsymbol{\alpha})$ から多項分布 $\mathbf{p} = (p_1, \cdots, p_K)$ を生成し, $\mathbf{p}$ からランダムにカテゴリをサンプリングして, $Y = \{2815, 507, 17, 4603, \cdots\}$ のようなデータを作ってみましょう. これから $\mathbf{p}$ を推定したとき, 式(2.1)の最尤推定と, 式(3.47)のディリクレ平滑化では真の値 $\mathbf{p}$ との差はどのようになるでしょうか. データ量が異なると, この

差はどう変わるでしょうか. なお, 確率 p_k を比較する際には 1.4 節で行ったように, 情報量 $-\log p_k$ に直して比較した方がよいでしょう.

[3–8] 適当なテキストを使って加算平滑化, ディリクレ平滑化 (階層ディリクレ言語モデル), Kneser–Ney 平滑化を用いた言語モデルを学習し, テストデータのパープレキシティを計算して, よい言語モデルほどパープレキシティが低くなっていることを確かめてみましょう.

[3–9] 同じテキストから PMI や GloVe で学習された単語ベクトルは, Word2Vec で学習されたものと, 類似語についてどのような違いがあるでしょうか. 学習された単語ベクトル同士の違いを, どのように定量化すればよいでしょうか. なお, 語彙が同じであれば, V 個の単語のそれぞれについて, V 個の単語の番号を類似度が高い順に並べると, その全体は $V \times V$ の行列で書くことができます. ただし, 単語の出現確率には大きな違いがあることに注意してください.

[3–10] 単語ベクトルの次元を変えたとき, 結果はどう変わるでしょうか. 単語ベクトルに必要な次元は, 行うタスクによって違うことが知られています. 複数の異なるタスクを使って, 次元による性能の違いを調べてみましょう.

[3–11] 単語ベクトルの中心化・白色化を行うと, 同様にして単語ベクトルの類似度がどのように変わるか計算してみましょう. こうした単語ベクトルの「性能」は, どのようにして評価したらいいでしょうか. 5.4.4 節で議論するトピックモデルの評価方法も参考にしてください.

[3–12] PMI 行列から SVD で計算できる単語ベクトルは, Word2Vec のように確率的勾配法によるわけではないため, 学習中に「引っ張られる」わけではありません. PMI による単語ベクトルの大きさは, どのように分布しているでしょうか.

[3–13] 実際の言語では単語は有限ではなく, 3.2 節でみたようにつねに新しい単語が出現します. こうした新語に対する単語ベクトルを計算する方法として, アラカルト埋め込み [112] という方法があります [*106]. アラカルト埋

*106　アラカルト (à la carte) とは, フランス料理において決まったプリフィクスのコースではなく, メニュー (carte) から選んでその場で注文する料理をさします.

め込みでは, 新しい語の周辺に現れた語の単語ベクトルを足し合わせ, そこからの線形回帰によって未知の単語ベクトルを予測します. [112]に従って回帰モデルを学習し, 新語の振る舞いを予測できるか調べてみましょう.

3 章の文献案内

3 章から, 本格的な自然言語処理に入りました. 言語や同様のシステムにみられる巾乗則などの統計的法則は, 『言語とフラクタル: 使用の集積の中にある偶然と必然』[113]で詳しく論じられています. 言語のように相互作用する複雑なネットワークについては, Barabási らによる[69]が統計力学からの古典的な導入です. 言語モデルとして LSTM [114]や Transformer [115] (2017 年) が使われるようになったのはごく最近で, 音声認識および機械翻訳を中心に n グラムモデルが長い間研究されてきました. Chen と Goodman によるハーバード大学のテクニカルレポート[89]は, さまざまな平滑化に関するわかりやすい入門で, 最も性能のよい Modified Kneser-Ney 平滑化はこの中で提案されています. その後, Goodman が 2001 年に出版した[116]は, ベイズ学習による[59]を除いては n グラム言語モデルの集大成で, 効率的な実装なども大変参考になるものです.

単語ベクトルを含む深層学習手法一般については, 自然言語処理の第一線の大学教員による『自然言語処理の基礎』[117]で解説されています. 英語では 3.5.4 節に登場した Goldberg による[118] (和訳[119]), 深層学習一般を扱った [120] (和訳[121]) などが知られています. 2017 年に現れた Transformer が含まれていないという大きな問題がありますが, 理論や考え方についてはより詳しく説明されています.

本章の後半でみたように, 単語ベクトルの数学的な取り扱いには, どうしても線形代数の知識が必要になります. 高松による 2020 年刊の『応用がみえる線形代数』[122]は, 線形代数の実例や使い方についても常に見据えつつ学習できる, 優れた入門です. 本格的な学習には, 線形代数の第一人者である MIT のストラング教授による 1978 年の『線形代数とその応用』[104]が概念の理解を重視した, 古典的な定番です [*107]. 応用を含んだ英語の入門には, 最適化で有名なスタンフォード大学の Boyd 教授による “Introduction to Applied Linear Algebra”(2018 年) [123]のような教科書もよいでしょう.

単語ベクトルの使用は自然言語処理だけでなく, 他の分野にも広がっています. 社会科学への応用については, 5 章の文献案内をご覧ください.

---

*107　「ストラングの線形代数」シリーズはその後でさまざまな版が出版されていますが, この本が最も基本的なものです.

5　文書の統計モデル

これまで, 文字・単語・文の順にテキストの単位と, その統計的なモデル化についてみてきました. 実際に私たちが出会う場面では, 単語だけ, 文だけが問題となることは少なく, 文が集まった**文書**を扱うことが多いでしょう. ここでいう文書とはいわゆる書類だけでなく, 普段触れる Web ページや電子メール, アンケートの自由回答, SNS 上の記事なども含まれます. 小説や論説, 法案といったものも, 時に非常に長くなりますが, 文書の一つといっていいでしょう.

文書としてのテキストの特徴は, **意味的まとまり**を持っているということです [*1]. たとえば, 図 5.1(a) のように完全に無関係な言葉の集まった文書は, まず存在しないでしょう [*2]. 実際の文書は図 5.1(b) のように, 何かのテーマ (ここでは「クラシック音楽」でしょう) に沿った内容が書かれているはずです. これはテキストが, 何かの意図を持って情報を伝える媒体であることから明らかでしょう. それでは, この文章の「意味的まとまり」をどうやって数学的に表現したらよいのでしょうか.

ケニア, 農学, 慶応, ラム, 希少, 帰還, パレスチナ, 損害, 栃木, 難病

(a) 単語がまったくランダムに選ばれた文書

奏者, 作曲, 歌曲, 弦, オペラ, 連作, 交響, 指揮, 鍵盤, 室内, ヴァイオリン

(b) 内容に意味的まとまりがある文書

図 5.1: 単語集合表現 (Bag of Words) と意味的まとまりの様子. □が単語の入った袋 (bag) を表しています.

∗1　これには本書で扱うもの以外に, 文体 (スタイル) 上のまとまりも含まれます. 一つの文書の中では一般にスタイルが統一されていることを利用して, 単語の埋め込みベクトルを意味の成分とスタイルの成分に分けて学習する興味深い研究に, 東北大の赤間らによる[179]があります.

∗2　3 章での, 意味を考えない n グラムモデルからの出力を思い出してみましょう.

5.1　ナイーブベイズ法と単語集合表現

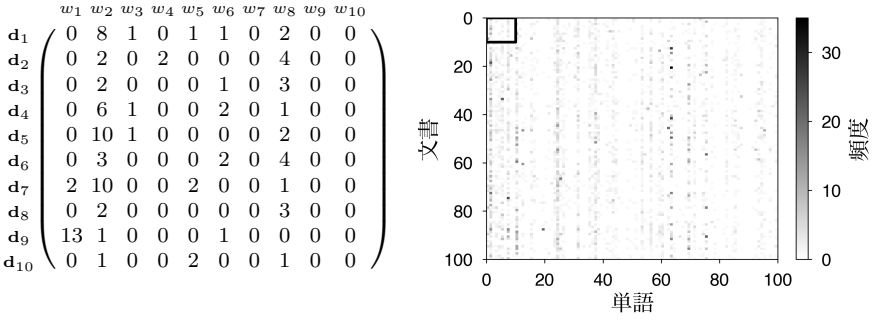
文書の意味的まとまりとして, 最も簡単なのは文書の「ジャンル」[*3], または分野を表すカテゴリといったものでしょう. 新聞やニュース記事には「社会面」「家庭欄」といった区別があり, それぞれ特定の言葉を使って, 一定のカテゴリの内容が書かれているのが普通です. 身近なところでは, 毎日届く迷惑メールも一種のカテゴリで, この場合, メールは「通常のメール」と「迷惑メール」の二種類のカテゴリに分けられることになります. また, 購買サイトのレビューなども主に, 「肯定的」および「否定的」なものに分けることができるでしょう.

Amazon での 5 億件を超える多言語のレビューのデータセット [*4] は公開されており[180], 日本語版だけ抜き出したものも Hugging Face で公開されています [*5]. 一方で, 新聞記事のようにカテゴリの付与された文書は有料のことが多く [*6], 特に日本語で自由に使用できるコーパスは少ないのですが, [181]で公開されている livedoor ニュースコーパスは, 表 5.1 に示した 9 個のカテゴリの記事からなる 7,367 文書の公開データです. たとえば「家電チャンネル」の記事には “売れ筋”, “加湿” といった言葉が, 「MOVIE ENTER」の記事には “公開”, “ヒーロー” といった言葉が多く出現するため, テキストを見れば, 9 つのカテゴリのどれに属する文書なのかが判定できそうです. なお以下では, 3.1 節で紹介した方法などで, 日本語の文書もあらかじめ単語に分けられていると仮定し

表 5.1: livedoor コーパスのカテゴリ名と本書で用いるカテゴリ番号, および文書数.

y	カテゴリ	カテゴリ名	文書数
1	MOVIE ENTER	`movie-enter`	870
2	独女通信	`dokujo-tsushin`	870
3	Sports Watch	`sports-watch`	900
4	IT ライフハック	`it-life-hack`	870
5	livedoor HOMME	`livedoor-homme`	511
6	Peachy	peachy	842
7	家電チャンネル	`kaden-channel`	864
8	エスマックス	`smax`	870
9	トピックニュース	`topic-news`	770

∗3　言語学では, レジスター (言語使用域) ともよばれています.
∗4　`https://amazon-reviews-2023.github.io/`
∗5　`https://huggingface.co/datasets/SetFit/amazon_reviews_multi_ja`
∗6　新聞記事コーパスの入手については, 85 ページを参照してください.



(a) 文書 1～10, 単語 1～10 の様子. (b) 文書 1～100, 単語 1～100 の様子.

図 5.2: livedoor コーパスでの文書–単語行列の一部. 横軸の単語は, コーパス全体での頻度順に並んでいます. この行列のうち 99.1%=123,721,815/124,833,815 は 0 で, 非常に疎な行列になっています. (b) の黒枠の中を拡大したものが (a) の行列です.

ます.

単語集合表現 こうした文書の意味内容を表す最も簡単な方法は, 文書の中で単語の順番を考えず, 頻度だけを数えることです. というのは, 言語には一般に非常に多くの語彙があるため, どんな単語が何回出現したのかを見れば, 文書の大まかな意味はほぼ特定できるからです [*7]. 図 5.1 のように, これは文書を単語の集合 (袋詰め) にしたものと考えることができるため, **単語集合** (Bag of Words) 表現とよばれています. 2 章で学んだ用語を使えば, これは単語が (文書ごとに異なる) ユニグラム分布から生成された, と仮定していることになります. [*8]

この場合, データは図 5.2 に示したように, 文書番号を縦に, 単語を横にとった**文書–単語行列**で表すことができます. この行列の (d, w) 要素が, d 番目の文書に単語 w が出現した回数 $n(d, w)$ になっています. 図からもわかるように, こ

∗7 これに対して, たとえば DNA では文字が ATGC の 4 種類しかありませんから, 各文字の出現頻度よりも, その並び方の方がはるかに重要になります.

∗8 「テキストを精密にモデル化する」という観点からは, これはずいぶん簡略化された表現です. 実際に, 文書内の単語の順番も考える DocNADE [182]のような方法も提案されており, パープレキシティではより低くなりますが, あまり使われているとはいえません. モデル化とは現実をそのまま再現することではなく, 「ある観点で見た」切り口を考えることで, 現象に関する理解を深め, 制御を可能にすることです[52]. これはむやみに複雑なモデルを使えばよいわけではない, ということの一つの現れといえるでしょう.

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	ラベル	y
$\mathbf{d}_1$	1		1		2		1	通常メール	0
$\mathbf{d}_2$		1	2		1	1			
$\mathbf{d}_3$	1	1		1		2		迷惑メール	1

図 5.3: ナイーブベイズ法での文書–単語行列の例. 空欄の部分は頻度が 0 であることを表しています.

の行列はほとんどが 0 で, 非常に疎 (スパース) な行列になっています. よって, 実際に表す際にはゼロでない要素だけを使って,

```
1:5 2:7 7:1 12:4 ...
```

のように, 3.4.3 節でも使った SVMlight 形式で表すとよいでしょう. `w:c` のような表記は, 単語 w が c 回出現したことを表します. つまりこの文書には, 単語 1 が 5 回, 単語 2 が 7 回, 単語 7 が 1 回, …出現したことを意味します. これは単語集合表現ですから, 順番には特に意味はありません.

たとえば, 最も簡単な例として, 文書–単語行列が図 5.3 のようになっているとしましょう. 文書 $\mathbf{d}_1$ と $\mathbf{d}_2$ は通常のメール ($y=0$), 文書 $\mathbf{d}_3$ は迷惑メール ($y=1$) というラベル y がわかっていたとします. 単語 w_1 や w_3 は day や work などの普通の単語, w_4 や w_6 は cash や dollar といった迷惑メールによく現れる単語を表していると考えてください.

このとき, 通常メール ($y=0$) での単語の確率分布は, 最も簡単には $\mathbf{d}_1$ と $\mathbf{d}_2$ での頻度を合計して総和で割ればよく,

$$p(w|y=0) = \left(\frac{1}{10}, \frac{1}{10}, \frac{1+2}{10}, \frac{0}{10}, \frac{2+1}{10}, \frac{1}{10}, \frac{1}{10}\right) = (0.1, 0.1, 0.3, 0, 0.3, 0.1, 0.1) \tag{5.1}$$

となります. 一方, 迷惑メール ($y=1$) の場合は文書は $\mathbf{d}_3$ の 1 つだけなので,

$$p(w|y=1) = \left(\frac{1}{5}, \frac{1}{5}, \frac{0}{5}, \frac{1}{5}, \frac{0}{5}, \frac{2}{5}, \frac{0}{5}\right) = (0.2, 0.2, 0, 0.2, 0, 0.4, 0)$$

です. また, $y=0$ の文書は $\mathbf{d}_1$ と $\mathbf{d}_2$ の 2 個, $y=1$ は $\mathbf{d}_3$ の 1 個ですから,

$$p(y) = \left(\frac{2}{3}, \frac{1}{3}\right) = (0.67, 0.33) \tag{5.2}$$

になります.

実際のテキストでは, $p(y)$ や $p(w|y)$ はどうなっているでしょうか. サポートサイトにある livedoor.py を, livedoor コーパス ldcc-20140209.tar.gz を展開したフォルダの下にある text フォルダに適用して

```
% livedoor.py 展開したフォルダ/text livedoor.txt
```

を実行すると, 内部で MeCab で単語分割を行って URL などを除き, 結果を

```
topic-news　園山 真希 絵 が 経営 する 料理 店 を 閉店 、 その 経緯 に 非難 の 声 29 日 、 料理 研究 家 の 園山 真希 絵 が 、 自身 が 経営 する 家庭 料理 「 園山 」 を ...
dokujo-tsushin 友人 代表 の スピーチ 、 独 女 は どう こなし て いる ? もうすぐ ・ と 呼ば れる 6 月 。 独 女 の 中 に は 自分 の 式 は まだ な のに ...
```

のように, [ラベル]<TAB>単語列.. の形で 1 文書が 1 行のテキスト livedoor.txt に書き出します. この livedoor.txt に対して, $p(y)$ を求めるスクリプト nblabels.py を実行してみると,

```
% nblabels.py livedoor.txt
⇒ dokujo-tsushin  0.1181
  it-life-hack    0.1181
  kaden-channel   0.1173
  :
  sports-watch    0.1222
  topic-news      0.1045
```

のようになりました. また, 式(5.1)のように $p(w|y)$ を計算する nbprob.py を使って, “Peachy” カテゴリ ($y=6$) での単語分布 $p(w|y=6)$ を求めると,

```
% nbprob.py livedoor.txt peachy
⇒ 、        -> 0.047520
  の        -> 0.046050
  に        -> 0.029309
  :
  私        -> 0.000718
  クリスマス -> 0.000716
```

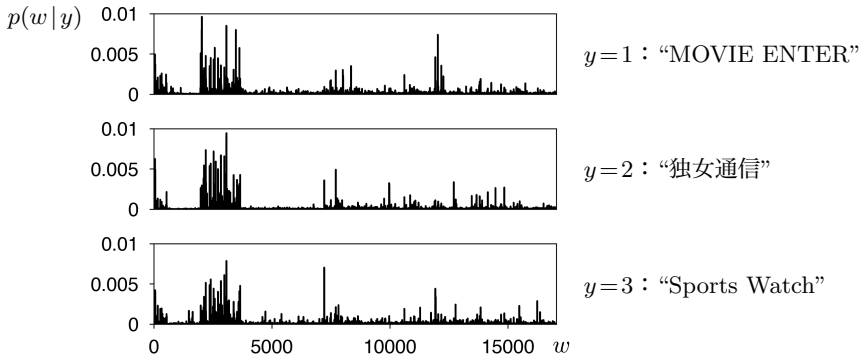


図 5.4: livedoor コーパスにおけるカテゴリ別の単語確率分布 $p(w|y)$ の様子. 横軸の単語は, 辞書順に並んでいます. “が”, “の” など極端に確率の高い語があるため, 確率が 0.01 以上の単語はプロットから除外しています.

```
恋愛　　-> 0.000709
:
乗れ　　-> 0.000002
考慮　　-> 0.000002
```

のように計算することができます. このカテゴリ別の単語確率分布 $p(w|y)$ の様子を, 図 5.4 に示しました.

5.1.1　文書の分類確率

これまでに求めた確率を使うと, 文書の確率を計算することができます. カテゴリ y のラベルを持つ文書 $\mathbf{d} = w_1 w_2 \cdots w_T$ の確率とは, $\mathbf{d}$ と y の同時確率のことですから, 式(2.20)の確率の連鎖則から

$$
\begin{aligned}
(5.3) \qquad p(\mathbf{d}, y) &= p(y)\, p(\mathbf{d}|y) \\
&= p(y) \prod_{w \in \mathbf{d}} p(w|y) \qquad (\textbf{ナイーブベイズ法の文書確率})
\end{aligned}
$$

となります. $p(\mathbf{d}|y)$ がユニグラム確率の積 $\prod_{w\in\mathbf{d}} p(w|y)$ に分解されることが, 単語集合の仮定に対応しています.

ノート：単語のカテゴリ所属確率の計算

上で計算したのはカテゴリ y の事前確率 $p(y)$ と, y から単語 w が出力される確率 $p(w|y)$ ですが, 実はこれから, w がカテゴリに所属する確率 $p(y|w)$ を求めることができます. というのは, 式(2.36)のベイズの定理から,

$$p(y|w) \propto p(w|y)p(y) \tag{5.4}$$

となるからです. 式(5.4)の右辺を和が 1 になるように正規化すれば, 単語 w の各カテゴリ y への所属分布 $p(y|w)$ が得られます.
livedoor コーパスで実際に計算してみると, 表 5.2 のようになりました. 確かに, 各カテゴリの特徴がよく表れていることがわかります. 「劇場」のように, ほぼ特定のカテゴリにしか所属しない単語にどんなものがあるか, 調べてみると面白いでしょう (→演習 5–2)

カテゴリ	劇場	恋愛	Mac	携帯	ゴルフ	肌
movie-enter	87.9	8.4	0.0	1.5	0.3	1.5
dokujo-tsushin	1.0	47.5	0.2	10.5	0.6	13.7
sports-watch	0.0	2.3	0.0	1.9	11.7	1.7
it-life-hack	0.0	0.3	83.4	15.4	0.9	0.3
livedoor-homme	0.8	2.3	4.0	7.0	81.4	2.9
peachy	3.6	29.9	0.0	3.1	1.2	75.0
kaden-channel	1.7	2.4	8.9	32.1	2.1	4.2
smax	1.5	0.2	3.5	22.3	1.1	0.1
topic-news	3.5	6.7	0.0	6.2	0.8	0.6

(%)

表 5.2: livedoor コーパスのナイーブベイズ法で, 単語がカテゴリーに所属する事後確率 $p(y|w)$ の例. わかりやすいよう, 確率を 100 倍して%で示しています.

たとえば図 5.3 のデータで, 新しい文書 $\mathbf{d} = w_2w_2w_5w_7$ がカテゴリ $y=0$ から生成される確率は,

$$\begin{aligned} p(\mathbf{d}, y=0) &= p(y=0) \prod_{w \in \{2,2,5,7\}} p(w|y=0) \\ &= 0.67 \times (0.1 \times 0.1 \times 0.3 \times 0.1) = 0.0002 \end{aligned}$$

になります.

このように文書の確率が計算できると, 新しい文書 $\mathbf{d}$ がどのカテゴリに属するかを, 確率的に予測できるようになります. これはつまり, 確率分布

$$p(y|\mathbf{d}) \qquad (y \in \{1, \cdots, K\}) \tag{5.5}$$

が知りたいということですから, 式(2.30)のベイズの定理を用いれば

$$p(y|\mathbf{d}) \propto p(y)\, p(\mathbf{d}|y) = p(y) \prod_{w \in \mathbf{d}} p(w|y) \tag{5.6}$$

となり, この確率は $y \in \{1, \cdots, K\}$ について式(5.3)から求めることができるからです.

実際に計算してみましょう. 図 5.3 のデータで $\mathbf{d} = w_1 w_2 w_6$ のとき, $p(y|\mathbf{d})$ は

$$p(y|d) \propto p(y)\, p(\mathbf{d}|y) = \begin{cases} 0.67 \times 0.1 \times 0.1 \times 0.1 \\ 0.33 \times 0.2 \times 0.2 \times 0.4 \end{cases} = \begin{cases} 0.00067 & (y=0) \\ 0.00528 & (y=1) \end{cases} \tag{5.7}$$

となります. よって, これを和が 1 になるように正規化して (2.4.2 節),

$$p(y|\mathbf{d}) = \left(\frac{0.00067}{0.00067+0.00528},\ \frac{0.00528}{0.00067+0.00528} \right) = (0.113,\ 0.887) \tag{5.8}$$

が求める分布となります. つまり $\mathbf{d} = w_1 w_2 w_6$ は $y=1$, すなわち迷惑メールである確率が 0.887 と非常に高い, ということがわかります.

このように, 式(5.3)の文書確率をもとにベイズの定理から, 式(5.6)を用いて文書を属するカテゴリに確率的に分類する手法を, **ナイーブベイズ法**といいます. ナイーブ (素朴) とは, テキストの単語の順番を考えない単語集合の仮定のことをさしていますが, この単純な仮定にもかかわらず, この後でみるように, 分類については高い性能を持つことがわかっており, 文書分類の最も基本的なモデルになっています. ここでは $p(w|y)$ を式(5.1)のように単に割り算して確率を計算しましたが (最尤推定), こうすると 3.4 節で学習したように, 式(5.7)でどれかの確率が 0 になると, 全体の確率が 0 になってしまいます. よって, もっとも簡単には頻度に小さな値 α を足して, 式(3.47)のようにディリクレ平滑化を行うとよいでしょう. すなわち, ナイーブベイズ法のパラメータの計算は, 次

のようになります. N 個の文書 $\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_N$ のカテゴリを $y_1, y_2, \cdots, y_N$ としたとき, $n(i,w)$ を文書 $\mathbf{d}_i$ で単語 w が出現した頻度とすると,

$$(5.9)\quad \begin{cases} p(y=k) \propto \sum_{i=1}^{N} \mathbb{I}(y_i=k) \\ p(w|y=k) \propto \sum_{i=1}^{N} \mathbb{I}(y_i=k)\big(n(i,w)+\alpha\big) \end{cases} \qquad (\textbf{ナイーブベイズ法の学習})$$

となります. 右辺を和が 1 になるように正規化すれば, 左辺の確率が得られます. ここで指示関数 $\mathbb{I}()$ は, 217 ページでも説明したように () の中が成り立つとき 1, そうでなければ 0 をとる関数です. 原理的には $p(y)$ の計算にも平滑化を適用できますが, 頻度が 0 のカテゴリはないと考えてよいため, こちらは最尤推定でもよいでしょう.

なお, 式(5.7)のような確率は一般に単語数が増えると, 確率が指数的に小さくなり, そのままでは計算機で表現できなくなってしまいます. よってこうした場合は, 対数をとって

$$(5.10)\quad \begin{aligned} \log p(y=0|\mathbf{d}) &\propto \log p(y) + \log p(\mathbf{d}|y) \\ &= \log(0.67) + \log(0.1) + \log(0.1) + \log(0.1) \\ &= -0.405 - 2.303 - 2.303 - 2.303 = -7.314 \end{aligned}$$

と計算するのがよいでしょう. このとき, $\log p(y)$ および $\log p(w|y)$ のそれぞれを「スコア」とみなせば, 式(5.10)は文書 $\mathbf{d}$ が $y=0$ に属する「スコア」を, ラベル y および $\mathbf{d}$ に含まれる各単語 w ごとに足し込んでいることになります. この合計スコアが最も大きいカテゴリが, 分類の結果になります. [*9]

ナイーブベイズ法の実験

実際に, livedoor コーパスで実験してみましょう. 先に作成したデータファイル `livedoor.txt` を, 2 章の 2.6 節で説明したように行をシャッフルして, ランダムに 80%の学習データ, 10%の開発データ, 10%のテストデータに分割します.

```
% split.py livedoor.txt livedoor
```

*9 一般に, 確率的でないアルゴリズムで天下りに定義された「スコア」は, 確率的に考え直すと, 確率の対数 (=情報量) とみなせることが非常に多くみられます.

これにより, livedoor.train (5888行), livedoor.dev (736行), livedoor.test (743行) が作成されます. ランダムに分割するため, 以下の数値は人によって多少異なることに注意してください.

同じ場所にある nb.py を使って, 学習データからナイーブベイズ法のモデルを計算します. この計算は数秒で終了します.

```
% nb.py livedoor.train nb.livedoor
N = 5888 docs, V = 17053 vocabs, K = 9 classes.
alpha = 0.01, threshold = 10.
saving model to nb.livedoor.. done.
```

nb.livedoor はナイーブベイズ法のパラメータ, つまり $p(y)$ および $p(w|y)$ が格納された gzip 圧縮済みの pickle ファイルです. 紙面の都合でスクリプトは載せませんので, 必ず中身を読んでから使ってください.

学習したモデルを使って新しいテキストのカテゴリを予測するには, スクリプト nbinf.py を使って, 次のように実行します.

```
% cat nb.txt
-　　誕生 日 に ディナー で プレゼント を もらっ た ！
% nbinf.py nb.livedoor nb.txt
- [ 0.086  0.028  0.000  0.000  0.012  0.872  0.001  0.000  0.001] 誕生日にディナーでプレゼントをもらった！
```

nb.txt は livedoor.txt と同じ形式で, 先頭の “-” はこのテキストのラベルが未知であることを表しています. 表5.1より, このテキストは6番目の “Peachy” カテゴリの確率が高い (0.872) と予測されたことがわかります. なお, 次に確率の大きい (0.086) カテゴリは “MOVIE ENTER” でした.

特定のテキストではなく, モデル全体の性能を評価するには, スクリプト nbeval.py を使って, livedoor.test のカテゴリの予測精度を次のようにして測ります [*10].

```
% nbeval.py nb.livedoor livedoor.test
accuracy = 91.92%
```

このテストデータでの文書カテゴリの予測精度は, 91.9%になりました.

結果の検証と「正解」ラベル　ということは, 8.2% (60/743 文書) はカテゴリの予測を「間違った」ということになります. ただし, この 8.2%がどんな場合

*10　ここでは開発データ livedoor.dev は使いません.

表 5.3: livedoor コーパスでナイーブベイズ法が「間違えた」文書の例.

テキスト	予測ラベル	正解ラベル
川面を流れるの美しさに感動!散った桜が川面を埋め尽くす光景のあまりの美しさが話題に先週、花見客のマナーの悪さを紹介した「これからお花見の..	peachy	it-life-hack
五輪サッカー英韓戦を前に韓国では「最悪のシナリオ」の声 2 日、韓国のニュースサイト「」は、五輪サッカー男子で韓国がとの予選リーグ最終試合を..	sports-watch	topic-news
オトナ女子たちの圧倒的支持をうけ、ドラマ 10『』の一挙再放送が決定!昨年を出産した女優の木村佳乃さんが 2 年ぶりに連ドラ主演を務め、現在 NHK ..	movie-enter	dokujo-tsushin

なのかには注意が必要です. 実際に予測を「間違った」60 件の文書を調べてみると, 表 5.3 のようになっており, 少なくとも人間の基準では, それほど間違いとはいえないことがわかります. 一方で, BERT のようなブラックボックスのニューラル手法は, このデータについて 96%程度の正解率を持っていますが [*11], これらは同じサッカーの記事でも, 「トピックニュース (一般向けのニュース)」と分類しているわけです. これはもしかするとニューラル手法が, livedoor コーパスの各分野のライターの書き癖を学習しており, 内容に関係なく文体で分類しているのかもしれません (少なくとも, その可能性があります). したがって, 「誤り」とされたものが本当に誤りなのかは, こうして結果を確認して検討した方がよいでしょう. この結果は, 人手による「正解」ラベルが本当に常に正しいのか, を教えてくれる例ともいえます.

テキストの感情極性分類 もう一つ, ナイーブベイズ法のわかりやすい応用に**感情分析**があります. 感情分析には, テキストを肯定的・否定的などの**極性**に分類する感情極性分類 (sentiment analysis) と, “期待”, “驚き” などの基本感情を付与する基本感情分類 (emotion detection) があり, 愛媛大学の梶原らは, 日本語の SNS テキスト (ツイート) に対して上記の感情極性と基本感情の強度を付

*11 `https://github.com/hppRC/bert-classification-tutorial` の結果によります.

表 5.4: WRIME コーパスのツイート極性から計算した, positive と negative それぞれのカテゴリ y での単語確率 $p(w|y)$ の上位語.

(a) y＝positive の場合

!	0.1060	♡	0.0730	ない	0.0930	怒ら	0.0800
♪	0.0890	好き	0.0720	“	0.0930	憂鬱	0.0800
最高	0.0850	フィンランド	0.0720	つらい	0.0880	無視	0.0800
楽しみ	0.0820	アニ	0.0710	嫌	0.0850	しんどい	0.0790
嬉しい	0.0810	ww	0.0710	しろ	0.0850	下痢	0.0790
曲	0.0800	^)/	0.0710	イライラ	0.0850	怒り	0.0790
かわいい	0.0770	Saint	0.0710	悪い	0.0850	寂しい	0.0790
サマ	0.0750	Snow	0.0710	くさい	0.0830	—	0.0790
おいしい	0.0740	さん	0.0710	吐き	0.0820	注意	0.0780
歌	0.0740	楽しかっ	0.0710	迷惑	0.0820	生理	0.0780
o	0.0740	きれい	0.0700	むり	0.0820	やらかし	0.0780
可愛	0.0730	*	0.0700	無理	0.0810	地獄	0.0780
可愛い	0.0730	＼(^	0.0700	なくなる	0.0810	吐き気	0.0780
良かっ	0.0730	しあわせ	0.0700	悲しい	0.0810	リスク	0.0780
ケーキ	0.0730	DVD	0.0700	めんど	0.0800	起き	0.0780

与したコーパス WRIME [183, 184]を公開しています [*12].

ここでは, より簡単な感情極性分類を行ってみることにしましょう. WRIME のデータをダウンロードし, 含まれる `wrime-ver2.tsv` にサポートサイトにある `wrime.py` を実行すると, SNS のテキストを 94 ページの `neologdn` で正規化した後で MeCab で単語に分割し, `livedoor.txt` と同じ形式のデータ (8740 行) を作ることができます. `neologdn` をインストールしていない方は, 先に `pip install neologdn` などでインストールしておいてください.

```
% git clone https://github.com/ids-cv/wrime
% cd wrime
% wrime.py wrime-ver2.tsv > wrime.txt
% shuf wrime.txt | head -3
negative        掃除 機 が 動か ない (^ q ^) なんて こっ た ..
positive        おはよう ござい ます ! 今日 は 学び に 使い ..
negative        22 時 に 閉まる すき家 って はじめ て み た
```

このうち, ランダムに選んだ 740 ツイートをテストデータ, 残りの 8000 ツイートを学習データとして, 先ほどと同様に `nb.py` でナイーブベイズ法のモデルを計算してみましょう. この計算は 1 秒未満で終わります.

*12 `https://github.com/ids-cv/wrime`

```
% shuf wrime.txt > wrime.shuffled.txt
% head -8000 wrime.shuffled.txt > wrime.train
% tail -740 wrime.shuffled.txt > wrime.test
% nb.py wrime.train nb.wrime
N = 8000 docs, V = 1690 vocabs, K = 2 classes.
alpha = 0.01, threshold = 10.
saving model to nb.wrime.. done.
```

計算したナイーブベイズ法の単語確率 $p(w|y)$ (式(5.3)) の上位語を, y が positive と negative の場合のそれぞれについて表 5.4 に示しました. 単語の極性が, きわめて自然な形で学習されていることがわかります. このモデルをもとに, テストデータのツイートの感情極性を予測してみましょう.

```
% nbinf.py nb.wrime wrime.test
⇒ 'negative': 0, 'positive': 1
negative [ 0.589  0.411] 手の痺れ一生治らん
positive [ 0.001  0.999] 先輩の息子が、まじ一瞬だけどつかまら..
positive [ 0.474  0.526] 私は、恐らく予定を組むのが好きな人な..
positive [ 0.464  0.536] チロルチョコにダイブしたい。
negative [ 1.000  0.000] わたしの伝え方がおかしいのか?なぜ、..
positive [ 0.739  0.261] 「落ちる」場面や、走っても走っても前..
positive [ 0.224  0.776] Shift+win キー+S キーで範囲指定の画面..
positive [ 0.091  0.909] おはようございます。
:
```

この場合, 数字は順に negative, positive の確率を表します. 短いツイートでは情報が少なく判断が難しくなりますが, ほぼ最初のカラムに示した「正解」のラベルに沿った予測ができていることがわかります. 図 5.5 に, 正解ラベルの Positive/Negative に分けた場合の Positive 確率の予測結果を示しました. 次のように実行して, ランダムな 20 ツイートの極性を予測値でソートしてみると, **非常に簡単なモデルにもかかわらず**, 確かにツイートの感情極性順に並んだ結果が得られることがわかります.

```
% nbinf.py nb.wrime wrime.test | shuf | head -15 | sort -k3 -n
⇒ positive [ 0.000  1.000] あー(*´∇`)キセキたちが私を萌殺そう..
positive [ 0.014  0.986] 明日、甥っ子に会う、甥っ子に会うう、..
positive [ 0.020  0.980] 椎名林檎かっこよかった
positive [ 0.046  0.954] ライブに向けて髭を伸ばしてきたのは、..
positive [ 0.096  0.904] 明日は歴史的な日ですぞ!
```

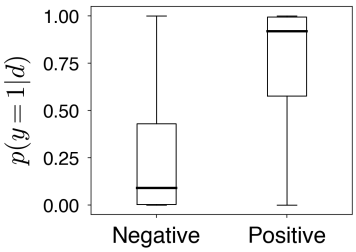


図 5.5: WRIME コーパスのテストデータのツイート極性の予測結果. 縦軸は, 予測された極性が Positive である確率を表します. 正解が Positive な場合には確率は 1 に近く, Negative な場合には 0 に近くなっており, 正しい学習が行えていることがわかります.

```
positive [ 0.156  0.844] あら、マクロス F の新譜売ってる
positive [ 0.250  0.750] 明日から三日間、東京出張だべさ
positive [ 0.294  0.706] 30 歳児の歯磨きで腹筋崩壊
negative [ 0.585  0.415] チャランネルのコメ欄閉じられてるは
positive [ 0.639  0.361] なんとなくこの曲知ってるけど歌詞詳し..
negative [ 0.664  0.336] ずいぶん昔からコカインやってたんだな..
negative [ 0.727  0.273] すきま風ってなんや…
positive [ 0.747  0.253] うおー肩と腰が痛いわーごみ出しに行か..
negative [ 0.941  0.059] 娘、一週間の咳と発熱。気管支炎の診断..
negative [ 0.999  0.001] トレード中にウイルスバスターのスキャ..
```

ノート：対数確率と logsumexp

式(5.8)の確率は式(5.7)の事後確率の和が 1 になるように正規化して求めたものですが, 文書の長さが大きくなると, この計算はそのまま行うことができません. というのは, 一定以上小さな確率は計算機では表すことができず, 0 になってしまうからです.[*13] よって, 式(5.10)のように対数をとって計算するのがよいでしょう.

K 個のカテゴリについて, 非常に小さな確率 $(p_1, p_2, \cdots, p_K)$ の対数 $(\ell_1, \ell_2, \cdots, \ell_K)$ がわかっていたとき, p を正規化して和を 1 にするには,

$$p'_k = \frac{p_k}{\sum_{k=1}^{K} p_k} = \exp\left(\ell_k - \log \sum_{k=1}^{K} \exp(\ell_k)\right) \tag{5.11}$$

で計算することができます. ただし上の式の $\log\sum_{k=1}^{K}\exp(\ell_k)$ は, そのまま計算すると, $\exp(\ell_k)$ (たとえば $\exp(-1000)$) がすべて 0 になってしまい, 求めることができなくなってしまう場合が多くあります.
このとき, $\ell_1, \ell_2, \cdots, \ell_K$ の中で最大のものを m とおけば,

$$\begin{aligned}(5.12)\quad \log\sum_{k=1}^{K}\exp(\ell_k) &= \log\left(e^{\ell_1}+e^{\ell_2}+\cdots e^{\ell_K}\right)\\ &= \log e^m\left(e^{\ell_1-m}+e^{\ell_2-m}+\cdots e^{\ell_K-m}\right)\\ &= m+\log\sum_{k=1}^{K}\exp(\ell_k-m) \qquad (\textbf{logsumexp})\end{aligned}$$

となります. こうすると ℓ_k の間の差だけが exp の中に残るため, 値がすべて 0 になるのを防ぐことができます. 式(5.12)で $\log\sum\exp()$ の値を計算する関数を, logsumexp といいます [49, §2.5.4]. Python では `scipy.special.logsumexp()` で使えるほか,

```
from numpy import exp,log
def logsumexp (x):
    y = max (x)
    return y + log(sum(exp(x - y)))
```

で定義すれば計算することができます.

5.2 ユニグラム混合モデル (UM)

前節で, ナイーブベイズ法が非常に簡単なモデル化と計算にもかかわらず, 文書を高い性能で確率的に分類できること, また, 人の付与した「正解」ラベルが必ずしも意味的な内容とは一致しないことをみてきました.

そもそも, テキストの内容が “トピックニュース” や “Peachy” といったラベルだけで語れるはずがなく, それぞれのカテゴリの中には, さまざまな話題が含まれているはずです. しかし, それらの「話題」が何かが, テキストに明示的に書かれているわけではありません. また一般のテキストには, そもそもラベルすらないことがほとんどです. 図 5.6 に示したのは, サポートサイトの本章のフォ

∗13 執筆時の手元の環境 (64bit) では, 倍精度実数の最小値は 2.385×10^{-277} でした.

```
<doc>
カート・ヴォネガット（Kurt Vonnegut、1922年11月11日 - 2007年4月11日）は、アメリカの小説家、エッセイスト、劇作家。1976年の作品『スラップスティック』より以前の作品はカート・ヴォネガット・ジュニア（"Kurt Vonnegut Jr."）の名で...
</doc>
<doc>
世界の放送方式（せかいのほうそうほうしき）高精細度テレビジョン放送（HDTV："High Definition Television"）に対して従来のテレビ放送の画質は標準テレビジョン放送（SDTV："Standard Definition Televisi on"）とも言われ、ここでは主に標準テレビに分類される方式について記述している。...
</doc>
<doc>
Mozilla Application Suite（モジラ・アプリケーション・スイート）または Mozilla Suite（モジラ・スイート）は Mozilla Foundation によりプロジェクトを組んでオープンソースで開発されていたインターネットスイートであり、...
</doc>
```

図 5.6: 日本語 Wikipedia からランダムに抽出した記事を集めた `jawiki.txt` の一部.

ルダにある `jawiki.txt` の一部で, これは日本語 Wikipedia の記事全体 (約 62 万記事) から筆者がランダムに 10,000 記事の概要部分を抜き出したテキストですが, これらの記事にはもちろんラベルはありません.

それでは, こうしたラベルのないテキストを扱うにはどうすればいいのでしょうか. 基本的な戦略は, 「**ラベルがないのだから, 自分で推定してしまえ**」ということです.

いま図 5.7 のように, 図 5.3 と同じデータでラベル y が未知だったとしましょう. この場合, 推定するのは人がつけたラベル y ではなく, 文書の潜在的なカテゴリを表す**潜在変数**ですので, 以後は y と区別して, z と書くことにします. 簡単のため, カテゴリの数を 2 とすると (後でもっと大きい場合も考えます), 各文書 $\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3$ についてカテゴリ $z \in \{0, 1\}$ がわからないのですから, (完全に対称だと以下の計算が進まないため) $(0.5, 0.5)$ から少しずらした確率分布

$$p(z|\mathbf{d}_1) = (0.4,\ 0.6), \quad p(z|\mathbf{d}_2) = (0.6,\ 0.4), \quad p(z|\mathbf{d}_3) = (0.4,\ 0.6)$$

を初期値として, 更新していくことを考えてみましょう [*14]. このとき $p(z)$ は,

*14 この確率をクラスタリングでは, 文書の各クラスタへの**負担率** (responsibility) といいます. ここでは, 負担率がほぼ等しい状態を初期値にして計算を始めています.

$z \in \{0,1\}$ について各文書がもつ確率の総和として

$$\begin{array}{ccc} & \mathbf{d}_1 & \mathbf{d}_2 \quad \mathbf{d}_3 \end{array}$$

$$(5.13) \qquad p(z) \propto \sum_{i=1}^{3} p(z|\mathbf{d}_i) = \begin{cases} 0.4+0.6+0.4=1.4 \\ 0.6+0.4+0.6=1.6 \end{cases} \propto \begin{cases} 0.467 & (z=0) \\ 0.533 & (z=1) \end{cases}$$

と求められます. これは, 式(5.9)で文書がどちらかのカテゴリに所属するとして, $\mathbb{I}()$ を使って 0/1 で数えていた頻度を, 確率に拡張してソフトな和をとったと言っていいでしょう. また $p(w|z)$ も, 同様に式(5.9)で頻度を z ごとに $\mathbb{I}()$ のかわりに確率 $p(z|\mathbf{d}_i)$ で重みづけて数え,

$$(5.14) \qquad p(w|z) \propto \sum_{i=1}^{N} p(z|\mathbf{d}_i)\, n(i,w)$$

で計算することができます.[*15] 実際に求めてみると, $z=0$ の場合は, 図 5.7 の行列を縦に読んで

$$\begin{cases} p(w_1|z=0) \propto 0.4\times 1 & & +0.4\times 1 = 0.8 \\ p(w_2|z=0) \propto & 0.6\times 1 & +0.4\times 1 = 1.0 \\ p(w_3|z=0) \propto 0.4\times 1 & +0.6\times 2 & = 1.6 \\ \quad\vdots \qquad\qquad \vdots & & \vdots \\ p(w_7|z=0) \propto 0.4\times 1 & & = 0.4 \end{cases}$$

となりました. これを総和が 1 になるように正規化し, $z=1$ の場合も同様に計算すると,

$$\begin{cases} p(w|z=0) = (0.114,\ 0.143,\ 0.229,\ 0.057,\ 0.200,\ 0.200,\ 0.057) \\ p(w|z=1) = (0.150,\ 0.125,\ 0.175,\ 0.075,\ 0.200,\ 0.200,\ 0.075) \end{cases}$$

が得られます.

すると, いま求めた $p(z), p(w|z)$ から, 式(5.3) すなわち

$$(5.15) \qquad p(z|\mathbf{d}) \propto p(z) \prod_{w\in\mathbf{d}} p(w|z)$$

---

*15 ここでは直感的な説明をしていますが, なぜこの計算をしてよいのかは, この後の 5.2.2 節で EM アルゴリズムを導入した際に説明します.

$$
\begin{array}{c} \\ \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 \end{array} \\
\left(\begin{array}{ccccccc}
1 & & 1 & & 2 & & 1 \\
 & 1 & 2 & & 1 & 1 & \\
1 & 1 & & 1 & & 2 &
\end{array} \right)
\end{array}
\begin{array}{c} z \\ ? \\ ? \\ ? \end{array}
$$

図 5.7: ユニグラム混合モデルでの文書–単語行列の例. 図 5.3 と同じデータですが, ここでは各文書のラベル z は未知で, 各テキストでの単語の頻度だけが与えられています. このとき, 文書 $\mathbf{d}_1$,$\mathbf{d}_2$,$\mathbf{d}_3$ について何がいえるでしょうか?

を使って, $p(z|\mathbf{d})$ をまた計算することができます. たとえば, 文書 $\mathbf{d}_2$ については

$$
\begin{array}{c}
\begin{array}{ccccc} z & & w_2 & w_3 & w_5 \quad w_6 \end{array} \\
\left\{ \begin{array}{l}
p(z{=}0|\mathbf{d}_2) \propto 0.467 \times 0.143 \times (0.229)^2 \times 0.200 \times 0.200 = 0.00014008 \\
p(z{=}1|\mathbf{d}_2) \propto 0.533 \times 0.125 \times (0.175)^2 \times 0.200 \times 0.200 = 0.00008162
\end{array} \right.
\end{array}
$$

から,

$$
p(z|\mathbf{d}_2) = (0.632, 0.368)
$$

となり, $p(z|\mathbf{d}_2)$ が $(0.6, 0.4) \to (0.632, 0.368)$ に更新されることがわかります. このように, 適当な初期値から始めて

$$
\left\{ \begin{array}{l}
p(z|\mathbf{d}) \text{ を推定する} \\
p(z), p(w|z) \text{ を推定する}
\end{array} \right. \tag{5.16}
$$

ことを繰り返すと, 表 5.5 に示したようにこの計算は 7 回程度の繰り返しで収束し, 結果として

$$
p(z|\mathbf{d}_1) = (1.000,\ 0.000),\ p(z|\mathbf{d}_2) = (1.000,\ 0.000),\ p(z|\mathbf{d}_3) = (0.000,\ 1.000)
$$

が得られます. **何も教えていないのに, 教師データ y があったときと同じ分類ができてしまいました!** $p(z|\mathbf{d}_1)$ は最初, $(0.4,\ 0.6)$ と $z{=}1$ の方が確率が高かったにもかかわらず, 式(5.16)の計算の繰り返しの中で逆転し, 正しい確率分布が得られていることに注意してください. なお, 実際の計算では 3 章で行ったように, $p(z)$ および $p(w|z)$ について平滑化を行った方がよいでしょう.

このように, 式(5.16)の計算を繰り返すことで, **教師なしでテキストをクラス**

表 5.5: ユニグラム混合モデルの学習過程. ここでは 7 ステップで学習が収束し, 教師なしで文書を 2 つの種類に分類できています.

文書	$\mathbf{d}_1$		$\mathbf{d}_2$		$\mathbf{d}_3$	
繰り返し	$z=0$	$z=1$	$z=0$	$z=1$	$z=0$	$z=1$
0(初期値)	0.400	0.600	0.600	0.400	0.400	0.600
1	0.399	0.601	0.630	0.370	0.367	0.633
2	0.423	0.577	0.668	0.332	0.303	0.697
3	0.519	0.481	0.715	0.285	0.183	0.817
4	0.769	0.231	0.776	0.224	0.047	0.953
5	0.982	0.018	0.873	0.127	0.002	0.998
6	1.000	0.000	0.988	0.012	0.000	1.000
7	**1.000**	**0.000**	**1.000**	**0.000**	**0.000**	**1.000**

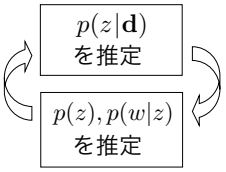
タリングする方法を, **ユニグラム混合モデル** (Unigram Mixtures, UM) [185]といいます. このアルゴリズムを, 図 5.8 に示しました. **混合モデル**とは, 各データが複数のモデル (この場合は $z=1,\cdots,K$ に対する単語分布 $p(w|z)$) のどれかから生成されており, 全体としてこの後の式(5.34)にみるように, 複数のモデルの確率的な混合になっていることをいいます [*16]. これまでの説明でわかるように, UM は「教師なしナイーブベイズ法」ともみなすことができ, 機械学習で K 平均法[36]として知られるクラスタリングを, テキストの多項分布の場合に適用したものになっています. なお, 実際の計算の際は式(5.9)の場合と同様に, $p(z)$ および $p(w|z)$ について平滑化を行った方がよいでしょう.

ユニグラム混合モデルの実験

実際の文書でも UM によるクラスタリングを行ってみることにしましょう. 本節の冒頭に示した `jawiki.txt`(図 5.6) SVMlight 形式のデータにするには, 同じフォルダにある `text2data.py` を次のように実行します. 最後の数字は, 単語頻度の閾値です. スクリプトの使い方は, 中身を読んでみてください.

```
   % text2data.py jawiki.txt jawiki 10
⇒  writing dic to jawiki.lex.. done.
   writing data to jawiki.dat.. done.
```

*16 式(3.64)のベイズ的な n グラムモデルも, こうした混合モデルになっているとみなすことができます.



EM アルゴリズムの概要. 上の E ステップと下の M ステップを, 収束するまで交互に繰り返します.

アルゴリズム 4: UM の EM アルゴリズム

パラメータ $p(z)$, $p(w|z)$ を適当に初期化する.
while 収束するまで **do**
  (* E ステップ *)
  for $i = 1 \cdots, N$ **do**
    $p(z|\mathbf{d}_i)$ を式(5.15)で計算する.
  (* M ステップ *)
  $p(z|\mathbf{d}_i)$ に基づいて, パラメータ $p(z)$, $p(w|z)$ を式(5.13)および式(5.14)で更新する.

図 5.8: ユニグラム混合モデルの推定法の概要と EM アルゴリズム.

これにより, 10000 行のデータ `jawiki.dat` と, 7509 個の語彙と ID の対照表 `jawiki.lex` が作られます.[*17] このデータについて, UM の実装 `um.py` を実行してみましょう.

```
   % um.py -K 100 jawiki um.jawiki.K100
⇒  UM: 10000 documents, 7509 words in vocabulary.
   iter[ 1] : PPL = 2580.88
   iter[ 2] : PPL = 1541.67
   iter[ 3] : PPL = 684.78
   iter[ 4] : PPL = 625.30
   iter[ 5] : PPL = 617.85
   iter[ 6] : PPL = 615.83
   iter[ 7] : PPL = 614.67
   iter[ 8] : PPL = 614.36
   converged.
   saving model to um.jawiki.K100.. done.
```

スクリプトの使い方は, `% um.py` をそのまま実行すれば表示されます. ここでは, $K = 100$ 個のカテゴリを仮定しました. 図 5.9 に示したように, 学習されたモデルで $p(z|\mathbf{d})$ を表示してみると, 日本語 Wikipedia の文章がさまざまなカテゴリに所属する様子が, 自動的に学習されていることがわかります.

*17 UM の場合は「が」「の」のような機能語を入れてしまうと, それに引っぱられてクラスタリングがうまく働かないため, ひらがなだけからなる単語を除くといった前処理を行っています.

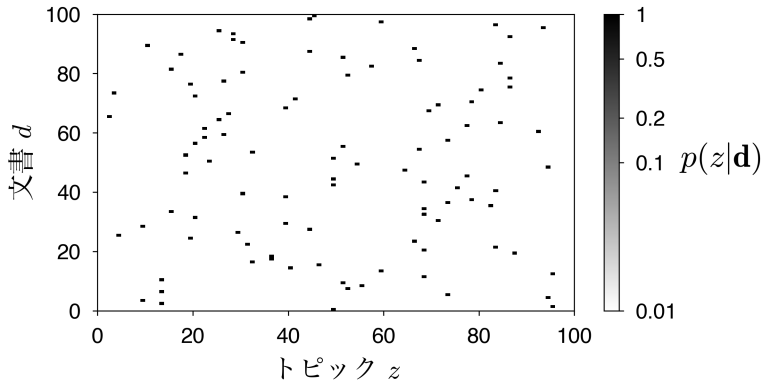


図 5.9: 日本語 Wikipedia コーパス `jawiki.txt` から学習された UM による, 各文書のトピックへの所属確率 $p(z|\mathbf{d})$ (最初の 100 文書). トピック数は $K = 100$ としました. ほとんどの $p(z|\mathbf{d})$ はほぼ 1 になっており, 文書ごとに異なるトピックに所属していることがわかります. このプロットは `umplot.py` で作ることができます.

5.2.1 トピックの解釈と自己相互情報量

ラベルのないテキストについて UM を学習することで, K 個のカテゴリ z ごとの単語の出力分布 $p(w|z)$ と, カテゴリの事前確率 $p(z)$ を学習することができました. この場合の z はナイーブベイズ法のように人が与えたカテゴリではなく, データから統計的に学習されたものですから, 一般にこれらを**トピック** (話題) とよびます. 教師なしナイーブベイズ法である UM は, 最も簡単な**トピックモデル**の一つです.

それぞれのトピック z は, 何を表しているのでしょうか. これは, 単語分布 $p(w|z)$ の様子を見てみればわかるはずです. ただし, この際には注意が必要です. 各トピック z について単に確率 $p(w|z)$ の大きい単語を表示すると, 表 5.6(a) に示したように “年” や “日” といった語が上位に来てしまい, トピックの意味がほとんどわからなくなってしまいます. これは, 考えてみると当然の現象です. UM では, 文書に含まれる語はすべて, あるトピック z から $p(w|z)$ に従って生成されたと考えています. すると, どの文書でも共通して現れる語は, どのトピックでも高い確率にならなければ, データをよく説明できないからで

す．ここでは “の” のような平仮名だけからなる語を前処理で除いていますが，除かない場合はこうした語が，すべてのトピックで出現確率の上位を占めることになります．[*18]

したがって重要なのは，単語 w のトピック z での生起確率 $p(w|z)$ 自体の大きさではなく，w の平均的な出現確率

$$p(w) = \sum_{z=1}^{K} p(w,z) = \sum_{z=1}^{K} p(w|z=k)\, p(z=k) \tag{5.17}$$

との比をとった，

$$\frac{p(w|z)}{p(w)} \tag{5.18}$$

の値でしょう．式(5.18)は，単語 w がトピック z で「通常より何倍高い確率で現れるのか」を表しています．“陶器” のように，全体的な確率は低くても，特定のトピックでは高い確率で現れる語は，$p(w|z)/p(w)$ の値は大きくなります．一方で “日” のように，出現確率は高くても，どのトピックでもほぼ同じ確率で現れる場合は，$p(w|z)/p(w)$ はほぼ 1 になると考えられます．よって，式(5.18) の対数をとって，

$$\mathrm{PMI}\,(w,z) = \log \frac{p(w|z)}{p(w)} \tag{5.19}$$

を計算すれば，“日” のような語ではほぼ $\log 1 = 0$，“陶器” のような語では > 0 になるでしょう．つまり，この形で単語の統計的な「重み」が得られます．表 5.6(b) に，`jawiki` コーパスについて式(5.19)で計算した値が大きい単語を各トピックについて示しました．これで，だいぶトピックの差がみえてきました．

式(5.19)を，w と z の**自己相互情報量** (Pointwise Mutual Information, PMI) といいます．というのは，式(2.30)のベイズの定理を用いれば，式(5.19)は

$$\log \frac{p(w|z)}{p(w)} = \log \frac{p(w,z)}{p(w)p(z)} \tag{5.20}$$

*18　この例からわかるように，言語では固定された「ストップワード」のリストを指定して，それらを除外すれば問題が解決するわけではありません．必ずこのように，リストには含まれない，同様に意味が薄い語が現れるからです．ある語がストップワードであるか否かは，ルールではなく，単語の振る舞いから統計的に判断すべき問題です．

表 5.6: 日本語 Wikipedia コーパスでの各トピックを表す特徴語の計算.

(a) 生成確率 $p(w|z)$ を使った上位語

トピック 1		トピック 10		トピック 20		トピック 30	
年	0.056	年	0.037	年	0.066	年	0.059
日	0.025	月	0.025	月	0.019	大学	0.027
月	0.024	日	0.017	日	0.019	月	0.026
音楽	0.022	システム	0.009	世	0.016	日本	0.021
作曲	0.019	社	0.009	王	0.012	日	0.019
曲	0.017	的	0.007	伯	0.010	会	0.013
家	0.016	法律	0.007	前	0.010	部	0.012
者	0.015	法	0.006	公	0.009	者	0.011
指揮	0.011	日本	0.006	紀元	0.008	長	0.010
作品	0.009	使用	0.006	語	0.007	委員	0.009

(b) PMI (w, z) を使った上位語

トピック 1		トピック 10		トピック 20		トピック 30	
大刀	4.502	inotify	4.538	ロジスティック	4.528	法科	4.079
査証	4.343	カーネル	4.418	τ	4.516	学長	3.625
ジュリアン	4.313	ボトル	4.292	ネブカドネザル	4.508	ラトビア	3.612
グレコ	4.282	ボランティア	4.055	マウイ	4.499	民兵	3.545
管弦	4.170	上杉	3.989	巨星	4.244	医科	3.495
奏	4.066	宝石	3.787	諸侯	4.211	土木	3.491
石巻	4.051	長尾	3.778	写像	4.199	商科	3.416
ヴァイオリニスト	4.038	帯域	3.775	伯	4.126	法学	3.413
長調	4.029	GNU	3.753	司馬	4.068	志願	3.353
初演	3.913	ペット	3.684	ブランデンブルク	4.055	黒川	3.306

(c) NPMI (w, z) を使った上位語

トピック 1		トピック 10		トピック 20		トピック 30	
作曲	0.462	カーネル	0.501	伯	0.476	工学	0.373
管弦	0.458	inotify	0.495	ロジスティック	0.457	大学	0.371
指揮	0.437	ボトル	0.490	τ	0.449	委員	0.368
ピアノ	0.437	ボランティア	0.463	ネブカドネザル	0.445	会長	0.353
交響	0.434	ペット	0.444	マウイ	0.441	理事	0.351
楽団	0.434	サーバ	0.438	ブランデンブルク	0.434	学会	0.347
大刀	0.434	帯域	0.430	諸侯	0.430	法学	0.345
グレコ	0.422	上杉	0.429	司馬	0.430	土木	0.332
音楽	0.422	Linux	0.427	辺境	0.428	長	0.327
協奏	0.421	宝石	0.425	写像	0.423	学長	0.327

と変形することができ, この値は情報理論で現れる, 確率変数 X と Y の間の相互情報量

$$I(X,Y) = \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{5.21}$$

で, Σ の中で期待値をとる対象の各要素 (pointwise) になっているからです.

正規化自己相互情報量　PMI を使うことで, 各トピックの意味はかなり明らかになりました. ただ, PMI の上位語にはまだ, 「ジュリアン」「ラトビア」などの特殊な単語が含まれてしまっています. これは PMI に一般にみられる現象で, 式(5.19)で分母に $p(w)$ があるため, $p(w)$ が小さい, すなわち稀な語であればあるほど, PMI の値が大きくなってしまいます.

この欠点を補うために, **正規化自己相互情報量** (Normalized PMI, NPMI) が 2009 年に提案されました［79］. 式(5.19)の PMI は, $p(w|z) = 1$, すなわち $p(w) = p(z)$ のときに最大値 $-\log p(w)$ をとることに注意しましょう. これから, w と z が完全に相関している, すなわち単語 w がカテゴリ z でしか現れない場合は, PMI は $p(w)$ が小さい稀な語ほど大きな値をとってしまう, ということがわかります.

よって, PMI をその最大値で割った

$$\log \frac{p(w|z)}{p(w)} \Big/ \big(-\log p(w)\big) \tag{5.22}$$

を考えることができ, これを正規化自己相互情報量 (NPMI) といいます. NPMI は $-1 \leq \mathrm{NPMI} \leq 1$ の値をとり,

- w と z が完全に相関しているとき 1,
- w と z が独立なとき 0,
- w と z が完全に逆相関しているとき -1

となり, w と z の相関を表すために理想的な量となっています.

なお, 式(5.22)は

$$\frac{\log p(w|z) - \log p(w)}{-\log p(w)} = 1 - \frac{-\log p(w|z)}{-\log p(w)} \tag{5.23}$$

とも書き直すことができ, 式(2.64)で学んだトピック z での単語 w の自己情報量 $-\log p(w|z)$ の, w の平均的な情報量 $-\log p(w)$ に対する比を最大値 1 から引いた値になっています. NPMI は一般には, 式(5.20)で最大値 $-\log p(w,z)$ $(=-\log p(w)=-\log p(z))$ で割って,

$$\text{NPMI}\,(w,z)=\log\frac{p(w,z)}{p(w)p(z)}\Big/\big(-\log p(w,z)\big) \tag{5.24}$$

(**正規化自己相互情報量**)

と定義されています[79].

表 5.6(c) に, NPMI で計算した UM の各トピックの特徴語を示しました. PMI と比べ, “太刀” や “ラトビア” といった低頻度語の順位が下がり, 各トピックの意味をより適切に表していることがわかります. こうした特徴から, NPMI はトピックモデルにおいて, トピックを説明する標準的な指標として広く使われています. なお, 表 5.6 ではまだ 1 つのトピックに複数の話題が混入してしまっていますが, この後で説明するように, これは UM をベイズ学習することで, 大きく改善することができます (表 5.7).

5.2.2 EM アルゴリズムによる学習

ところで, 図 5.8 の学習アルゴリズムはなぜ収束し, 教師なしで UM の学習ができるのでしょうか? それは, このアルゴリズムが **EM アルゴリズム**という一般的な学習法の一種になっているからです.

EM アルゴリズムは, 因果推論でも有名な Rubin らによって 1970 年代に提案され[186], 1990 年代には最先端の手法として機械学習で多く使われました [36]. クラスタリングによく使われる K 平均法など, 多くのアルゴリズムが EM アルゴリズムの一種とみなせる重要な方法ですので, 以下でみていくことにしましょう.

最尤推定とベイズ推定 まず, 統計モデルの目標は図 5.10 に示したように, データ $\mathcal{D}$ の背後にパラメータ θ があると仮定し, $\mathcal{D}$ を最もよく説明する θ を求めることです. このとき, θ から $\mathcal{D}$ が生成される確率 (=尤度) $p(\mathcal{D}|\theta)$ [*19] を最大に

*19 $\mathcal{D}$ は既知なので, これはパラメータ θ の関数となり, これを尤度関数とよぶのでした (39 ページ).

(a) 通常の統計モデル　(b) 潜在変数 z がある統計モデル

図 5.10: 統計モデルとパラメータの推定. $\mathcal{D}$ はデータを, θ はパラメータを表します.

する $\theta = \widehat{\theta}$ を求めること, すなわち

$$\widehat{\theta} = \operatorname*{argmax}_{\theta} p(\mathcal{D}|\theta) \tag{5.25}$$

を点推定するのが**最尤推定**です. いっぽう, 有限のデータから $\widehat{\theta}$ が一点で正確に求まるはずがありませんから, $\mathcal{D}$ が与えられた下での θ の**確率分布**

$$p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta)p(\theta) \tag{5.26}$$

を求めるのが**ベイズ推定**です. 式(5.25)と式(5.26)を見比べると, 最尤推定とは, θ の事前分布に一様分布 $p(\theta) \propto 1$ を仮定した上で, $p(\theta|\mathcal{D})$ をデルタ関数 $\delta(\theta=\widehat{\theta})$ で一点近似することに対応することがわかります. EM アルゴリズムは最尤推定を行う方法ですので, ここではしばらく式(5.25)で考えることにしましょう.

EM アルゴリズム　モデルにパラメータ θ だけでなく, UM の各文書のクラスタ番号のように**潜在変数** z があるとき, 式(5.25)の尤度は

$$p(\mathcal{D}|\theta) = \sum_{z} p(\mathcal{D}, z|\theta) \tag{5.27}$$

と書き換えることができます [*20]. これはつまり, データの確率は可能な z のすべての場合についての確率の和となることを意味しています.

実際には, きわめて小さな値になる式(5.27)の確率のかわりに, その対数をとった (これを**対数尤度**といいます)

$$\log p(\mathcal{D}|\theta) = \log \sum_{z} p(\mathcal{D}, z|\theta) \tag{5.28}$$

---

*20　z が連続値の場合は, 和のかわりに積分を使うことになります.

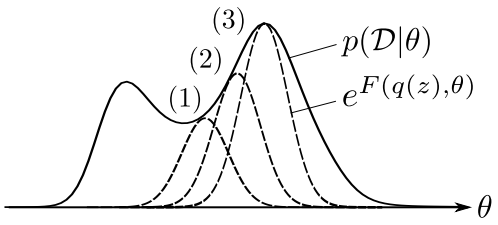


図 5.11: EM アルゴリズムによる学習. 実際には対数尤度 $\log p(\mathcal{D}|\theta)$ を最大化しますが, わかりやすさのため, もとの確率密度 $p(\mathcal{D}|\theta)$ の空間で示しています. EM アルゴリズムでは $\log p(\mathcal{D}|\theta)$ の下界 $F(q(z),\theta)$ を, 適当な初期値から始めて, E ステップと M ステップを繰り返すことで次々と最大化します. 初期値によっては, 左側の局所解に陥ることもあることに注意してください.

を最大化したいのですが, この式は log の中に積ではなく和が入っているため, $\log p()$ の形に分解することができません. そこで, 一つ工夫をします. 分母・分子に同じ値 (補助分布) $q(z)$ をかけて, 付録 C (345 ページ) の Jensen の不等式を使えば, 式(5.28)の対数尤度は

$$\begin{aligned}(5.29)\qquad \log p(\mathcal{D}|\theta) &= \log \sum_z p(\mathcal{D}, z|\theta) = \log \sum_z q(z)\,\frac{p(\mathcal{D}, z|\theta)}{q(z)} \\ &\geq \sum_z q(z) \log \frac{p(\mathcal{D}, z|\theta)}{q(z)} \;\equiv F(q(z), \theta)\end{aligned}$$

と, 不等式で下から抑えることができます. 補助分布 $q(z)$ を用いることで, 無事, $\log p(\mathcal{D}, z|\theta)$ の形に変形することができました！

式(5.29)は真の対数尤度 $\log p(\mathcal{D}|\theta)$ の下界ですから, 図 5.11 に示したように式(5.29)を θ および $q(z)$ について最大化することで, 真の対数尤度に下から近づくことができます.[*21] これを行うのが EM アルゴリズムです. 具体的には,

E ステップ ： θ を固定して, $F(q(z),\theta)$ を $q(z)$ について最大化する
M ステップ： $q(z)$ を固定して, $F(q(z),\theta)$ を θ について最大化する

を交互に繰り返すことで下限を逐次最大化します. E は Expectation (期待値), M は Maximization (最大化) を意味します. それぞれ, どんな計算になるでしょ

*21 この $F(q(z),\theta)$ を, 物理学とのアナロジーで自由エネルギーともいいます.

うか. やや抽象的ですが, 先に一般論を説明します. UM の場合の具体例は, その後でみてみることにしましょう.

E ステップ $q(z)$ については, 下限 $F(q(z),\theta)$ は

$$
\begin{aligned}
(5.30)\quad F(q(z),\theta) &= \sum_z q(z)\log\frac{p(\mathcal{D},z|\theta)}{q(z)} = \sum_z q(z)\log\frac{p(z|\mathcal{D},\theta)p(\mathcal{D}|\theta)}{q(z)} \\
&= \sum_z q(z)\log\frac{p(z|\mathcal{D},\theta)}{q(z)} + \log p(\mathcal{D}|\theta) \\
&= \underbrace{-D(q(z)||p(z|\mathcal{D},\theta))}_{(\star)} + \log p(\mathcal{D}|\theta)
\end{aligned}
$$

となることに注意しましょう. $(\star)$ の部分は式(2.70)で学んだ, $q(z)$ と $p(z|\mathcal{D},\theta)$ の KL ダイバージェンスですから, この値は

$$
(5.31)\qquad q(z) = p(z|\mathcal{D},\theta)
$$

のとき最小になり, よって F が最大になります. すなわち, $q(z)$ としては実際には, 現在のモデル (パラメータ θ) での z の予測分布 $p(z|\mathcal{D},\theta)$ をとればよい, ということがわかります.

M ステップ θ の方はどうでしょうか. 式(5.29)を θ について変形すると,

$$
\begin{aligned}
(5.32)\quad F(q(z),\theta) &= \sum_z q(z)\log\frac{p(\mathcal{D},z|\theta)}{q(z)} = \sum_z q(z)\log p(\mathcal{D},z|\theta) \\
&\quad - \sum_z q(z)\log q(z) = \Big\langle \log p(\mathcal{D},z|\theta)\Big\rangle_{q(z)} + H(q(z))
\end{aligned}
$$

です. $H()$ は式(2.66)のエントロピー関数で, $\langle\cdots\rangle_p$ は, $\cdots$ を p について期待値をとることを表しています. この式はこれ以上簡単になりませんので, F を θ について最大化するためには,

$$
(5.33)\qquad Q(\theta) = \Big\langle \log p(\mathcal{D},z|\theta)\Big\rangle_{q(z)}
$$

に対して, $\partial Q/\partial\theta = 0$ とおくなどして θ について最大化することになります. 式(5.33)は, 一般に **Q 関数**とよばれています. Q 関数は, 現在のモデルを使った潜在変数 z とデータの同時確率を, 未知の z について期待値をとったもので

あることに注意してください.

UM の EM アルゴリズムの導出

それでは, 具体的に計算してみましょう. UM で文書 $\mathbf{d}$ が生成される確率モデルは, 式(5.3) で y (ここでは z) について和をとった

$$p(\mathbf{d}) = \sum_{z=1}^{K} p(\mathbf{d}, z) = \sum_{z=1}^{K} p(\mathbf{d}|z)\, p(z) \tag{5.34}$$
$$= \sum_{z=1}^{K} p(z) \prod_{w \in \mathbf{d}} p(w|z) \qquad (\textbf{UM の文書確率})$$

となります. よって N 個の文書 $\mathcal{D} = \{\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_N\}$ 全体の確率は,

$$p(\mathcal{D}) = \prod_{i=1}^{N} \left(\sum_{z=1}^{K} p(z) \prod_{w \in \mathbf{d}_i} p(w|z) \right) \tag{5.35}$$

になり, 対数をとれば

$$\log p(\mathcal{D}) = \sum_{i=1}^{N} \log \left(\sum_{z=1}^{K} p(z) \prod_{w \in \mathbf{d}_i} p(w|z) \right) \tag{5.36}$$

です. パラメータは $\theta = \{\, p(z), p(w|z) \,\}_{z=1}^{K}$ になります.

E ステップ $p(z|\mathcal{D}, \theta)$ とはこの場合, 各文書 $\mathbf{d} \in \mathcal{D}$ について, その負担率 $p(z|\mathbf{d})$ のことです. これは, 式(5.15)で計算できるのでした.

M ステップ Q 関数の中身である $\log p(\mathcal{D}, z|\theta)$ は, この場合は

$$\log p(\mathcal{D}, z|\theta) = \log \left(p(z) \prod_{w \in \mathbf{d}} p(w|z) \right) = \log p(z) + \sum_{w \in \mathbf{d}} \log p(w|z)$$

になります. Q 関数はこの z に関する期待値なので, $q(z) = p(z|\mathbf{d})$ でしたから,

$$\left\langle \log p(\mathcal{D}, z|\theta) \right\rangle_{q(z)} = \sum_{z=1}^{K} p(z|\mathbf{d}) \log p(z) + \sum_{w \in \mathbf{d}} \sum_{z=1}^{K} p(z|\mathbf{d}) \log p(w|z) \tag{5.37}$$

となります. 文書は N 個あるので, 全体の Q 関数は

$$Q(\theta) = \sum_{i=1}^{N} \left[\sum_{z=1}^{K} p(z|\mathbf{d}_i) \log p(z) + \sum_{w \in \mathbf{d}_i} \sum_{z=1}^{K} p(z|\mathbf{d}_i) \log p(w|z) \right] \tag{5.38}$$

となります.

$p(z)$ について最大化： $Q(\theta)$ を $p(z)$ に関して微分すると[*22], $(\log x)' = \dfrac{1}{x}$ ですから

$$\frac{\delta Q(\theta)}{\delta p(z)} = \sum_{i=1}^{N} \frac{p(z|\mathbf{d}_i)}{p(z)} \tag{5.39}$$

です. $\sum_z p(z)=1$ なので, 制約項 $\left(\sum_z p(z)-1\right)$ を追加すると, ラグランジュの未定乗数法[*23] により, 未定乗数を λ として

$$\frac{\delta}{\delta p(z)} \left(Q - \lambda \left(\sum_z p(z) - 1 \right) \right) = 0 \tag{5.40}$$

を解けばよいことになります. よって式(5.39)から

$$\sum_{i=1}^{N} \frac{p(z|\mathbf{d}_i)}{p(z)} - \lambda = 0$$

となり, これを解いて,

$$p(z) = \frac{1}{\lambda} \sum_{i=1}^{N} p(z|\mathbf{d}_i) \tag{5.41}$$

となります. ここで λ についても微分すれば, 当たり前ですが

$$\sum_z p(z) = 1$$

となり, 式(5.41)で λ は和がちょうど 1 になるように決めればよいことがわかって, 式(5.13)が得られます.

$p(w|z)$ について最大化： $Q(\theta)$ を $p(w|z)$ について微分すると, 同様に

---

*22 $p(z)$ は関数なので, 形式的に変分 $\delta Q/\delta p(z)$ を用いていますが, Q の中に $p(z)$ の微分に関する項はありませんので, これは $p(z)$ に関する通常の偏微分になります[187]. $p(z=k)=\mu_k$ のようにおいて, 変数 μ_k について最大化すると考えてもよいでしょう.

*23 ラグランジュの未定乗数法について知らない方は, 紙幅の問題で本書では解説しませんので, [21, 1.2.3 節]などを参照してください.

$$\frac{\delta Q(\theta)}{\delta p(w|z)} = \sum_{i=1}^{N} \sum_{v \in \mathbf{d}_i} \mathbb{I}(v=w) \frac{p(z|\mathbf{d}_i)}{p(w|z)} = \frac{1}{p(w|z)} \sum_{i=1}^{N} p(z|\mathbf{d}_i)\, n(i,w) \tag{5.42}$$

となります. よって, 同様にラグランジュの未定乗数法により

$$\frac{1}{p(w|z)} \sum_{i=1}^{N} p(z|\mathbf{d}_i)\, n(i,w) - \lambda = 0 \tag{5.43}$$

から,

$$p(w|z) = \frac{1}{\lambda} \sum_{i=1}^{N} p(z|\mathbf{d}_i)\, n(i,w) \tag{5.44}$$

となり, 同様にして式(5.14)が得られました. □

EM アルゴリズムでは, E ステップと M ステップの繰り返しによって下限 $F(q(z),\theta)$ が単調に増加します. よって, 255 ページの実装 `um.py` のように学習データのパープレキシティを計算して, 減少が一定の閾値以下になったかどうかで収束を判定するとよいでしょう.

5.2.3* UM のベイズ学習

前節で, 図 5.8 のアルゴリズムが EM アルゴリズムになっていること, およびその導出について理解することができました. EM アルゴリズムでは, E ステップと M ステップを交互に行うことで, 真の対数尤度を図 5.11 のように下から近似して最大化します. これにより, 式(5.29)の $F(q(z),\theta)$ はつねに増加して極大値に近づきます. しかし, EM アルゴリズムは常に山登りを行うため, HMM の学習について図 4.25 でみたように, 初期値に依存して**一番近くの山 (局所解) につかまってしまう**という大きな欠点があります. 複雑な問題で対数尤度の曲面に凸凹が大きいほど, この問題は顕著になります.

これを解消するもっとも簡単な方法は, EM アルゴリズムの E ステップで負担率を計算して期待値をとるのではなく, この確率に従ってランダムにサンプリングを行うことでしょう. 常に山を登るのではなく, 時には下ることも許せば, より広い空間を探索することができます. 具体的には, UM の場合は E ステップでは負担率 $p(z|\mathbf{d}_n)$ に従ってトピック z_n を文書ごとに毎回サンプリングし,

M ステップではその z_n をナイーブベイズ法における「正解」ラベルと同様にみなして, パラメータ $\theta=\{p(z),p(w|z)\}$ を更新します. E ステップをサンプリングで置き換えるこの方法を, **モンテカルロ EM アルゴリズム** [188]といいます [*24].

モンテカルロ EM アルゴリズムは局所解から脱出できる有用な方法ですが [*25], M ステップでは θ について最大化を行っているため, 全体にみると EM アルゴリズムの逐次最大化の一種となっており, 真の最大値を探索することはできません. 局所解に陥らず, パラメータの真の最適解を求めるためには, θ についても最尤推定ではなく, 式(5.26)の事後分布全体を用いた**ベイズ学習**を行う必要があります.

幸いにして, UM の場合は 4 章で学習した **Gibbs サンプリング**を容易に行うことができます. 各文書 $\mathbf{d}_1,\cdots,\mathbf{d}_N$ について, その潜在トピック $\mathbf{z}=z_1,\cdots,z_N$ を $p(z|\mathbf{d}_i)$ に従ってサンプリングしたとしましょう. このとき, $\mathbf{z}$ の中でトピックが k となった文書の数を $n(k)=\sum_{i=1}^{N}\mathbb{I}(z_i=k)$ とおけば, $\boldsymbol{\mu}=\{p(z)\}$ $(z=1,\cdots,K)$ が事前分布

$$\boldsymbol{\mu}\sim \mathrm{Dir}(\alpha,\cdots,\alpha) \tag{5.45}$$

に従っているとしたとき, その事後分布は式(4.40)と同様に

$$\boldsymbol{\mu}|\mathbf{z}\sim \mathrm{Dir}(\alpha+n(1),\cdots,\alpha+n(K)) \tag{5.46}$$

となります. $\boldsymbol{\phi}_k=\{p(w|k)\}$ についても, トピック $z_n=k$ となった文書だけを集めて, その中での各単語 w の頻度 $m(k,w)$ を計算すれば, $\boldsymbol{\phi}_k$ が事前分布

$$\boldsymbol{\phi}_k\sim \mathrm{Dir}(\beta,\cdots,\beta) \tag{5.47}$$

に従っているとき, 事後分布は式(4.49)とまったく同様に,

$$\boldsymbol{\phi}_k|\mathbf{z}\sim \mathrm{Dir}(\beta+m(k,1),\cdots,\beta+m(k,V)) \tag{5.48}$$

---

*24 モンテカルロとはカジノで有名な地中海の都市で, 乱数を使って期待値を求める方法をこの名前にしたのは, 第二次世界大戦中に原爆開発のために作られたアメリカのロスアラモス研究所で, 数学者のウラムとフォン・ノイマンによるものです[189].

*25 モンテカルロ EM アルゴリズムは, 潜在変数 z に指数的な組み合わせの可能性があるなど, すべての場合を計算して期待値をとることが実質的に不可能な場合にも有益なアルゴリズムです.

アルゴリズム 5: UM の周辺化 Gibbs サンプリング

for i in $1, \cdots, N$ **do** (* 初期化 *)
  $z_i = \text{randint}(K)$ (* $1 \cdots K$ の乱数で初期化 *)
  $n'(z_i)$++
  for w in $\mathbf{d}_i$ **do**
    $m'(z_i, w)$++
while 収束するまで **do** (* Gibbs サンプリング *)
  for i in randperm(N) **do** (* $1 \cdots N$ をシャッフル *)
    $n'(z_i)$−− (* カウントを減らす *)
    for w in $\mathbf{d}_i$ **do**
      $m'(z_i, w)$−−
    z_i を式(5.15)および式(5.50)に従ってサンプリング
    $n'(z_i)$++ (* カウントを増やす *)
    for w in $\mathbf{d}_i$ **do**
      $m'(z_i, w)$++
式(5.50)を用いて, $p(z)$, $p(w|z)$ の期待値を求める

図 5.12: 周辺化 Gibbs サンプリングによる UM のベイズ学習のアルゴリズム. −− はカウントを 1 減らすことを, ++ はカウントを 1 増やすことを表します.

となります. $\mathbf{z}$ をサンプリングした後で, $\boldsymbol{\mu}$ と $\boldsymbol{\phi}$ を式(5.46)および式(5.48)からサンプリングし, これを繰り返せば, パラメータの事後分布からサンプリングを行ってベイズ学習することができます.

理論的にはこれでベイズ学習を行うことができますが, 4.4.1 節で HMM について学習したように, この方法は V 次元のパラメータ $\boldsymbol{\phi}_k$ を毎回サンプリングすることで, 推定値の分散が大きくなってしまうという問題があります. そこで HMM と同様に**周辺化 Gibbs サンプリング**を行えば, $\boldsymbol{\mu}$ および $\boldsymbol{\phi}$ を積分消去し, $\mathbf{z}$ **だけ**をサンプリングして効率的に学習を行うことができます. いま, i 番目の文書 d_i のトピック z_i をサンプリングしたいとしましょう. 周辺化 Gibbs サンプリングでは, まず $\mathbf{z}$ をランダムに初期化します. 次に, i 番目の文書のトピック z_i を, それ以外の文書のトピック $\mathbf{z}_{-i} = z_1, \cdots, z_{i-1}, z_{i+1}, \cdots, z_N$ を条件とした事後分布 $p(z_i|d_i, \mathbf{z}_{-i})$ からサンプリングして推定します. $\mathbf{z}_{-i}$ がわかっているのですから, 式(5.46)および式(5.48)において, 文書 d_i の分を除いた頻度を $m'(k, v)$ および $n'(k)$ とおけば, それぞれの事後分布は, 同様に

表 5.7: UM のベイズ学習で得られたトピックの特徴語 (NPMI). ここではトピック数 $K=100$ として学習しました. 表 5.6 の EM アルゴリズムによる学習と比べて, 局所解に陥らず, さらによいトピックが学習されていることがわかります. トピックの順番は実行によって毎回異なりますので, 対応しているわけではありません.

トピック 1		トピック 10		トピック 20		トピック 30	
アキレウス	0.643	南極	0.686	党	0.431	オリンピック	0.663
アステロパイオス	0.638	度	0.651	議員	0.387	団	0.635
ヒッポトオーン	0.633	西経	0.648	民主	0.385	北京	0.583
母音	0.619	北極	0.647	選挙	0.380	アテネ	0.582
ギリシア	0.615	東経	0.645	内閣	0.375	ロンドン	0.581
トロイア	0.607	線	0.623	大臣	0.368	バルセロナ	0.573
省略	0.598	成す	0.621	政治	0.365	結果	0.530
神話	0.574	角度	0.617	衆議	0.356	名簿	0.523
ゼウス	0.560	点	0.591	議会	0.344	競技	0.521
槍	0.548	通過	0.585	政党	0.343	選手	0.519

$$\text{(5.49)} \qquad \begin{cases} \boldsymbol{\mu} \,|\, \mathbf{z}_{-i} \sim \mathrm{Dir}\,(\alpha + n'(1), \cdots, \alpha + n'(K)) \\ \boldsymbol{\phi}_k \,|\, \mathbf{z}_{-i} \sim \mathrm{Dir}\,(\beta + m'(k,1), \cdots, \beta + m'(k,V)) \end{cases}$$

となります. この期待値は, ディリクレ分布の期待値の公式(3.30)から

$$\text{(5.50)} \qquad \begin{cases} \mathbb{E}[\mu_k \,|\, \mathbf{z}_{-i}] \;\; = \dfrac{\alpha + n'(k)}{\sum_{k=1}^{K} (\alpha + n'(k))} & \big(\, \mu_k = p(z_i = k) \,\big) \\ \mathbb{E}[\phi_{kv} \,|\, \mathbf{z}_{-i}] = \dfrac{\beta + m'(k,v)}{\sum_{v=1}^{V} (\beta + m'(k,v))} & \big(\, \phi_{kv} = p(v \,|\, z_i = k) \,\big) \end{cases}$$

となります. 式(5.50)を式(5.15)の $p(z_i \,|\, \mathbf{d}_i)$ の計算に用いれば, 文書 $\mathbf{d}_i$ のトピック z_i をサンプリングすることができます. これを, すべての $i = 1, \cdots, N$ について繰り返します. 以上を行う UM のベイズ学習 (周辺化 Gibbs サンプリング) のアルゴリズムを, 図 5.12 に示しました.

式(5.50)で期待値をとっているため, パラメータ $\theta = \{\, p(z), p(v \,|\, z) \,\}$ の点推定値はもはや存在しないことに注意してください. 各文書のトピック割り当てである $\mathbf{z}$ だけをサンプリングすれば, θ の分布は式(5.48)および式(5.46)の形でディリクレ事後分布として表されますので, その期待値を式(5.50)のように求めることができます. この方法は, パラメータの事後分布からサンプリングを

行う完全な MCMC 法となっているため, 局所解の危険がなく, 理論的には十分長くサンプリングを行えば, 図 4.25 のように真の最適解 (を含む事後分布全体) からのサンプルを得ることができます.

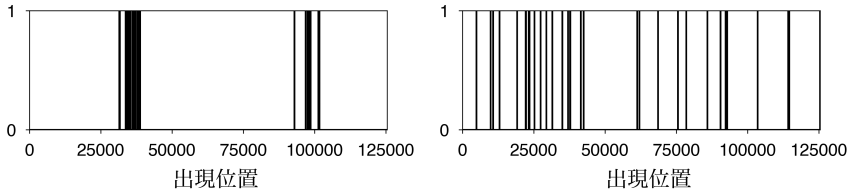
表 5.7 に, サポートサイトにある UM のベイズ学習の実装 `bum.py` を用いて, 同じ日本語 Wikipedia コーパスについて $K=100$ のトピック数で学習した結果を示しました. ここでは, 100 回の Gibbs サンプリング (MCMC) の繰り返しでモデルを学習しています. EM アルゴリズムと比べて, さまざまな可能性を確率的に試すことで, 意味的にさらにまとまりの高い潜在トピックが学習されていることがわかります. こうして, われわれは文書のクラスタリングの問題を, 完全にベイズ的に解くことができました. 1.5 節で触れたように, 適当に作った文書ベクトルを K 平均法でクラスタリングする, といった簡単な方法では, こうした精密なクラスタリングやトピック分布を学習することはできません.

5.3 ディリクレ混合モデル (DM)

ここまで, テキストの単語が式(5.3)のように多項分布に従って発生すると仮定してきました. この多項モデルでは, 確率 $1/100=0.01$ の単語が 2 回発生する確率は $(1/100)^2=1/10000=0.0001$ になります. しかし実際には, これはあまり正しくありません. 図 5.13 に, 『シャーロック・ホームズの冒険』の本文の中で特定の単語が出現した場所を示しました. これを見ると, “lestrade” (レストレード警部) は一度出現すると続けて何度も出現しており, これはそんなに低い確率ではなさそうです. 言葉の出現のこうした性質を, **バースト性**といいます. ただし, バースト性の度合いは単語によって異なり, 同じ頻度 (=確率) の “interest” は, ほぼ一様に出現していることがわかります. [*26]

バースト性の度合いは, 次のようにしても調べることができます[191]. いま, コーパスの各文書を前半と後半に分け, 2 章で行ったように前半を ‘train’ データ, 後半を ‘test’ データとよぶことにしましょう. 表 5.8 に, livedoor コーパスの 7367 文書の中で, “ソフトバンク” が前半と後半それぞれで 1 回以上出現した

*26 Altmann ら[190]は, こうした単語の出現間隔を故障時間の分布としても知られるワイブル分布でモデル化することで, ワイブル分布の形状パラメータ $0<\beta<1$ で単語の性質が表現できる, という興味深い研究を行っています.



(a) “lestrade”　(b) “interest”

図 5.13: 『シャーロック・ホームズの冒険』での “lestrade” と “interest” の出現位置. この 2 つの単語の出現頻度はどちらも同じ 38 回ですが, “lestrade” (レストレード警部) のような言葉は, テキストの中で一部にバースト的に出現しています.

表 5.8: livedoor コーパスの中で「ソフトバンク」が出現した文書の数. x はそこで単語が出現したことを, $\bar{x}$ は出現しなかったことを表します.

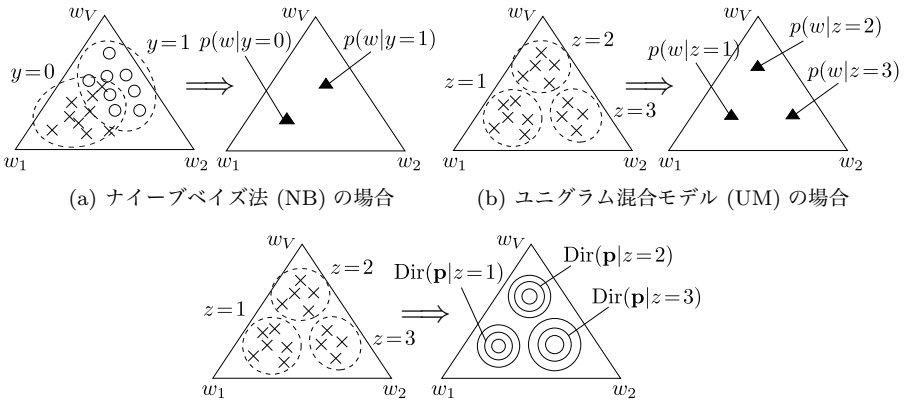
	test	$\overline{\text{test}}$
train	$a = 108$	$b = 93$
$\overline{\text{train}}$	$c = 93$	$d = 7073$

／しなかった文書の数をまとめました.[*27] a は両方出現した, b は前半のみ出現し後半にはなかった, c は前半にはなかったが後半に出現した, d は前半と後半どちらにも出現しなかった文書の数をそれぞれ表しています.

“ソフトバンク” の文書レベルでの出現確率は $(a+b+c)/(a+b+c+d) = 0.04$ ですので, 前半に出現した中 $(a+b)$ で, 後半にも出現する a の確率は, 単語の出現が独立ならば 0.04 程度になるはずです. しかし, この場合 $a/(a+b) = 0.537$ となっており, これは非常に高い確率です. つまり, “ソフトバンク” は 1 回出現すれば, 同じ文書でもう 1 回以上出現する確率は 0.537 もあるということです. これは明らかに, 単語のバースト性を示しています. [191]で (当時) ベル研究所の Church は, 出現確率 p の低い “Noriega”[*28] を例にとって 「“Noriega” が 2 回出現する (Two Noriegas) 確率は, p^2 ではなく $p/2$ だ」という言葉で, この現象を経験的に指摘しました. われわれの例でも, “ソフトバンク” が 2 回出現する確率は確かに, $p \cdot 0.537 \simeq p/2$ になっていることがわかります.

もちろん, バースト性は単語によって異なり, たとえば “順調” では $a/(a+b)$

*27　これは, サポートサイトにある `recur.py` で調べることができます.
*28　1980 年代にパナマの独裁者として有名だったノリエガ将軍 (1934–2017) のこと.



(a) ナイーブベイズ法 (NB) の場合　(b) ユニグラム混合モデル (UM) の場合

(c) ディリクレ混合モデル (DM) の場合

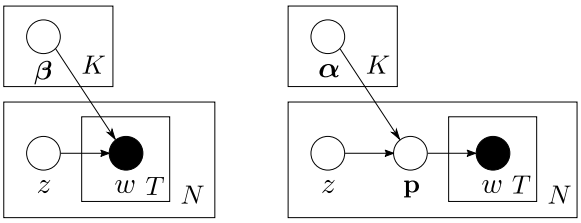
図 5.14: 文書モデルの幾何的表現. すべての多項分布は, $w_1 \cdots w_V$ を頂点とする単語単体の中に存在しています. NB や UM では, それらをトピックの代表点 (▲) に針の立った δ 関数で表しますが, DM ではそれぞれのトピックをディリクレ分布で表現します.

は 0.065 と低い確率でした. テキストの確率モデルは, こうしたバースト性を表現できると望ましいでしょう.

5.3.1 単語単体と幾何的解釈

ここで, 少し違う視点からナイーブベイズ法と UM について振り返ってみましょう. 5.1 節の単語集合の仮定から, 各文書には含まれる単語を生成した V 次元の確率分布 $\mathbf{p} = (p_1, p_2, \cdots, p_V)$ があることになります. 3.4.1 節でみたように, この $\mathbf{p}$ は図 5.14 の各図の左側のような, 各単語 $w_1, w_2, \cdots, w_V$ を頂点とする V 次元の単体の中に存在しています. これを, **単語単体**といいます. ナイーブベイズ法や UM は, この単語単体上に点としてトピック分布 $\mathbf{p} = \{p(w|y)\}$ や $\mathbf{p} = \{p(w|z)\}$ を, 図 5.14(a) や (b) のように学習するものだといえるでしょう. ナイーブベイズ法は UM の教師あり版ですから, 以下では UM に統一して考えていくことにします.

こうして考えると, UM のモデルは制限が強すぎるのではないでしょうか. 文書を生成したもともとの $\mathbf{p}$ にはさまざまな可能性があるにもかかわらず, UM では, トピック z ごとに特定の $\mathbf{p} = \{p(w|z)\}$ で代表してしまっています. 「単語の



(a) ユニグラム混合モデル (UM)　(b) ディリクレ混合モデル (DM)

図 5.15: UM と DM のグラフィカルモデル. ●は観測値, ○は確率変数を表します. 四角いプレートは繰り返しを, 右隅は繰り返しの数を表しています. DM では, 文書ごとに異なる $\mathbf{p}$ があるのが特徴です. 実際には, この $\mathbf{p}$ は積分消去することができます.

出やすさ」$\mathbf{p}$ のタイプが完全に固定されてしまっているので, むしろ, 図 5.14(c) のようにトピックごとに**分布**を考え, この分布から $\mathbf{p}$ が生まれたと考える方が適切ではないでしょうか. $\mathbf{p}$ 自体が多項分布なので, これは**確率分布の確率分布**となります.

この分布として最も簡単なのは, 3.4.1 節のディリクレ分布を考えることでしょう. 図 5.14(c) のように, 単語単体上にトピックを表現する複数のディリクレ分布を考えるこのモデルを, **ディリクレ混合モデル** (Dirichlet Mixtures, **DM**) といいます.[*29] DM は最初, バイオインフォマティクスの分野でアミノ酸配列の解析のために, 1996 年に提案されました[192]. 筑波大学の山本らは 2003 年に, これがテキストにも適用でき, 5.4 節で説明する LDA よりも高い文書確率を与えることを示しました[193].[*30]

ディリクレ混合モデルの生成モデル　UM では, 文書 $\mathbf{d} = w_1 w_2 \cdots w_T$ は次のようにして生成されたと考えました.

1. $z \sim \boldsymbol{\lambda}$ でトピック z を選択.
2. For $t = 1, \cdots, T$,
    a. $w_t \sim p(w|z)$ から単語 w_t を生成.

すなわち, 各単語はトピック z で決まる固定されたユニグラム分布 $\boldsymbol{\beta}_z = \{p(w|z)\}$

*29　これは, 通常のユークリッド空間において K 平均法の代わりに, ガウス混合モデルを考えることと似ています. ディリクレ分布は, 単体上のガウス分布のようなものと考えてよいでしょう.

*30　東京大学の佐藤らの研究[194]を用いれば, Pitman–Yor 過程を用いる洗練された方法で DM とこの後の LDA は統合でき, さらに高い文書確率を与えることが確かめられています.

から生成されると考えています. これをグラフィカルモデルで表すと, 図 5.15(a) のようになります. これに対して, DM では文書は

1. $z \sim \boldsymbol{\lambda}$ でトピック z を選択.
2. $\mathbf{p} \sim \mathrm{Dir}(\boldsymbol{\alpha}_z)$ でユニグラム分布 $\mathbf{p}$ を生成.
3. For $t = 1, \cdots, T$,
   a. $w_t \sim \mathbf{p}$ で単語 w_t を生成.

のようにして生成されたと考えます. グラフィカルモデルで表すと, 図 5.15(b) のようになります. UM では決まったユニグラム分布 $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ の中から選ぶのに対し, DM では**文書ごとに異なるユニグラム分布 $\mathbf{p}$** が, トピックに従って**毎回生成される**のが特徴です. これにより, DM ではたとえば同じ車の話題を扱う文書でも, 「トヨタ」が多い場合と「日産」が多い場合で異なる $\mathbf{p}$ が使われていると考えて区別することができます. 後で数学的に示すように, このことから単語のバースト性を説明することができます.

幸いにして $\mathbf{p}$ は直接求める必要はなく, 3 章で説明したように, 期待値をとって積分消去することができます. いま, 文書 $\mathbf{d}$ のトピックが $z=k$ とわかっていたとしましょう. すると文書の確率は, $\mathbf{p}$ が k 番目のディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha}_k)$ $(\boldsymbol{\alpha}_k = (\alpha_{k1}, \alpha_{k2}, \cdots, \alpha_{kV}))$ から生成されたということなので, 期待値をとれば 3.4.2 節で学んだように,

$$
\begin{aligned}
(5.51) \qquad p(\mathbf{d}\,|\,z=k) &= \int p(\mathbf{d}, \mathbf{p}\,|\,z=k)d\mathbf{p} = \int p(\mathbf{d}\,|\,\mathbf{p})p(\mathbf{p}\,|\,z=k)d\mathbf{p} \\
&= \int \prod_{t=1}^{T} \prod_{v=1}^{V} p_v^{\mathbb{I}(w_t=v)} \cdot \mathrm{Dir}(\mathbf{p}\,|\,\boldsymbol{\alpha}_k)d\mathbf{p} \\
&= \int \prod_{v=1}^{V} p_v^{n_v} \cdot \frac{\Gamma(\sum_v \alpha_{kv})}{\prod_v \Gamma(\alpha_{kv})} \prod_{v=1}^{V} p_v^{\alpha_{kv}-1} d\mathbf{p} \\
&= \frac{\Gamma(\sum_v \alpha_{kv})}{\Gamma(\sum_v \alpha_{kv} + L)} \prod_{v=1}^{V} \frac{\Gamma(\alpha_{kv} + n_v)}{\Gamma(\alpha_{kv})}
\end{aligned}
$$

となり, 式(3.55)の**ポリア分布**で表すことができます. ここで, 1 行目では同時確率の周辺化 (公式(2.10)) および確率の連鎖則 (公式(2.20)) を用いました. $n_v = \sum_{t=1}^{T} \mathbb{I}(w_t = v)$ は, $\mathbf{d}$ の中で単語 v が何回出たかを表します. 実際には z は未知ですから, DM での文書の確率は, z についても期待値をとり,

アルゴリズム 6: DM の EM-Newton アルゴリズム.

```
λ, α_1, ..., α_K を適当に初期化する
while 収束するまで do
    for i in 1, ..., N do                    (* E ステップ *)
        式(5.52) で p(z|d_i) を計算
    式(5.53) で λ, α_1, ..., α_K を更新       (* M ステップ *)
```

図 5.16: DM の EM アルゴリズムによる学習.

$$
\begin{aligned}
(5.52)\quad p(\mathbf{d}) &= \sum_{k=1}^{K} p(\mathbf{d}, z=k) = \sum_{k=1}^{K} p(\mathbf{d}\,|\,z=k)p(z=k) \\
&= \sum_{k=1}^{K} \lambda_k \frac{\Gamma(\sum_v \alpha_{kv})}{\Gamma(\sum_v \alpha_{kv}+L)} \prod_{v=1}^{V} \frac{\Gamma(\alpha_{kv}+n_v)}{\Gamma(\alpha_{kv})} \qquad (\textbf{DM の文書確率})
\end{aligned}
$$

となります.[*31]

ディリクレ混合モデルの学習　DM のパラメータは, $p(z)$ を表す $\boldsymbol{\lambda}=\{\lambda_1, \cdots, \lambda_K\}$ およびトピック別のディリクレ分布のハイパーパラメータ $\{\alpha_{kv}\,|\,k=1, \cdots, K, v=1, \cdots, V\}$ です. 他のトピックモデルも同様ですが, 一般に $K=100$, 語彙の大きさ $V=10000$ と少なめのときでも, α_{kv} は 100 万個程度のパラメータ数となることに注意してください. α_{kv} は EM アルゴリズムの中で式(3.57)のニュートン法で最適化することもできますが, これは特殊関数であるダイガンマ関数 $\Psi(x)=d/dx \log\Gamma(x)$ が含まれているため, テキストデータについては非常に重い計算になります.

筑波大学の山本ら[193]はこれにかわり, [83]で示された高速な LOO (Leave-One-Out) 近似を使って, 図 5.16 のアルゴリズムおよび次式で DM のパラメータを高速, かつ安定に学習できることを示しました.

$$
(5.53)\qquad \begin{cases} \lambda_k \propto \sum_{i=1}^{N} p(z_i=k\,|\,\mathbf{d}_i) \\ \alpha'_{kv} = \alpha_{kv} \cdot \dfrac{\sum_{i=1}^{N} p(z_i=k\,|\,\mathbf{d}_i)\, n(i,v) \,/\, (n(i,v)-1+\alpha_{kv})}{\sum_{i=1}^{N} p(z_i=k\,|\,\mathbf{d}_i)\, n(i) \,/\, (n(i)-1+\sum_v \alpha_{kv})} \end{cases}
$$

---

*31　すなわち,「ディリクレ分布の混合モデル」となっているのは $\mathbf{p}$ に対してのもので, $\mathbf{p}$ を積分消去すると, DM は観測値については「混合ポリア分布」になっています.

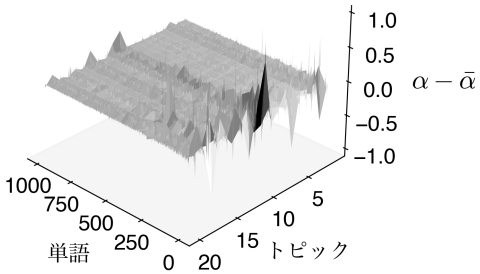


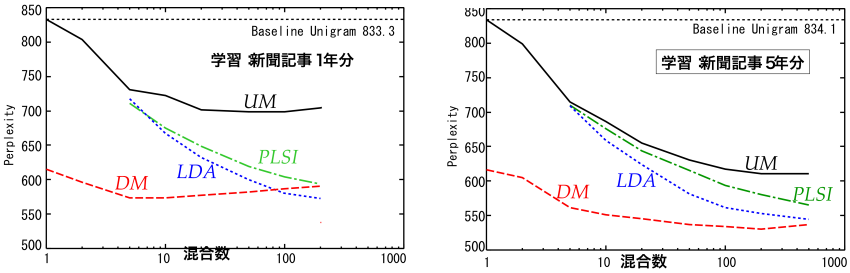
図 5.17: livedoor コーパスで学習された DM のパラメータ $\boldsymbol{\alpha}_1, \cdots, \boldsymbol{\alpha}_K$ の様子. 全体の平均 $\overline{\boldsymbol{\alpha}}$ との差分をプロットしています (単語 1〜1000, トピック数 $K = 20$).

ここで $n(i) = \sum_v n(i, v)$ は文書 i の長さを表します. この式は特殊関数を含まないため計算が速く, 筆者の公開している C 言語による実装 *32 を用いた場合, livedoor コーパスで $K = 100$ の場合は, 執筆時の環境では 3 分程度で EM アルゴリズムが収束します. Python では非常に遅くなるため, 注意してください. なお, この推定法は最尤推定のため過学習しやすく, 山本らはさらに, 階層ベイズの考え方と変分ベイズ法の一種で推定を安定化する Smoothed DM による解法を示し[195], こちらの方が安定して高性能であることが報告されています.*33 図 5.18 に, UM, DM および次節で説明する LDA を livedoor コーパスで学習した場合の, テストデータのパープレキシティをトピック数 K ごとに示しました. これからわかるように, パープレキシティ(文書確率) の面では, DM は LDA よりも優れたモデルです. ただしこれは主に, 文書内での単語のバースト性のモデル化から生じるもので, LDA の方が柔軟なモデルであることに注意してください.

なお, DM の各トピックを表すディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha}_k)$ の期待値は, 公式(3.30) で示したように $\boldsymbol{\alpha}_k$ の和を 1 に正規化すると求めることができ, これは UM におけるトピック分布 $p(w|z)$ と等価だとみなすことができます. DM ではどのようなトピックが推定されているか, 5.2.1 節の UM の場合と同様にして調べ

*32 `http://chasen.org/~daiti-m/dist/dm/`にある `dm-0.2.tar.gz` で C 言語による実装を公開しています.

*33 `http://chasen.org/~daiti-m/dist/dm/`にある `sdm-0.2.tar.gz` で C 言語による実装を公開しています.



(a) 学習データが少ない場合 (1 年分)　(b) 学習データが多い場合 (5 年分)

図 5.18: 各トピックモデルの性能 ([196]から引用). 横軸の混合数とはこの場合はトピック数のことで, 縦軸はテストデータに対するパープレキシティです. いずれの場合も, 単語のバースト性をとらえる DM の性能が最も高くなっており, 混合数を増やすと LDA の性能と近づいてくることがわかります.

てみると面白いでしょう (→演習 5–6).

5.3.2　ポリア分布と単語のバースト性

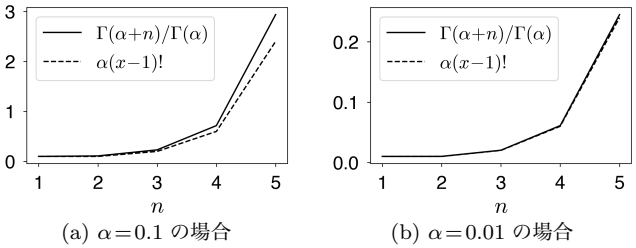
いま, 文書を生成したディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ のパラメータを $\boldsymbol{\alpha}=(\alpha_1,\cdots,\alpha_V)$ とすると, 式(5.52) に示したように, 文書 $\mathbf{d}$ の確率はポリア分布

$$p(\mathbf{d}) = \frac{L!}{\prod_{v\in d} n_v!}\frac{\Gamma(\sum_v \alpha_v)}{\Gamma(\sum_v \alpha_v + L)}\prod_{v\in d}\frac{\Gamma(\alpha_v+n_v)}{\Gamma(\alpha_v)} \tag{5.54}$$

となります. 長さ L の文書で各単語の頻度が n_v になる組み合わせの数は, 多項係数 $L!/\prod_v n_v!$ だけありますから, 上では明示的に加えました.

ここで, 3.4.3 節の図 3.18 でみたように, 言語の場合は α_v は非常に小さく, ほとんどの場合は 0.01 あるいは 0.001 未満の値となることを思い出してください. このとき, 式(5.54)で各単語に依存するファクター $\Gamma(\alpha_v+n_v)/\Gamma(\alpha_v)$ は, 式(3.37) の Γ 関数の性質から

$$\begin{aligned}\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} &= \frac{(n+\alpha-1)(n+\alpha-2)\cdots\alpha\cancel{\Gamma(\alpha)}}{\cancel{\Gamma(\alpha)}} \\ &= \alpha(\alpha+1)\cdots(n+\alpha-1) \\ &\simeq \alpha(n-1)! \qquad (\because \alpha \ll 1)\end{aligned} \tag{5.55}$$



(a) $\alpha=0.1$ の場合 (b) $\alpha=0.01$ の場合

図 5.19: ポッホハマー関数 $\Gamma(\alpha+n)/\Gamma(\alpha)$ の近似. α が小さい場合には, この値は近似 $\alpha(x-1)!$ と, ほとんど差がないことがわかります.

と近似することができます. 図 5.19 に, この近似が α が小さいとき, 実際に非常に正確であることを示しました. よって, この近似を式(5.54)に代入すれば,

$$p(\mathbf{d}) \simeq \frac{L!}{\prod_v n_v!}\frac{\Gamma(\sum_v \alpha_v)}{\Gamma(\sum_v \alpha_v + L)}\prod_{v\in d}\alpha_v(n_v-1)! \tag{5.56}$$

$$= L!\,\frac{\Gamma(\sum_v \alpha_v)}{\Gamma(\sum_v \alpha_v + L)}\prod_{v\in d}\frac{\alpha_v}{n_v} \qquad (\textbf{指数分布族 DCM 分布})$$

が得られます. n_v の項をくくり出せば, これは統計学で標準的な指数分布族の形で表すことができ, Elkan [197]はこれを指数分布族 DCM (EDCM) 分布と呼んでいます.

n	1	2	3	⋯
p	α_v	$\frac{\alpha_v}{2}$	$\frac{\alpha_v}{3}$	⋯

式(5.56)で, 文書中の単語 v に依存するファクターは α_v/n_v になっており, これは右の表のように, $n_v=1$ のときは確率が α_v 倍, $n_v=2$ のときは $\alpha_v/2$ 倍, ⋯ となることを意味しています. つまり, (頻度の低い) 単語 v が 2 回出現しても, その確率は α_v^2 ではなく, $\alpha_v/2$ 倍にしかならないということです. これはまさに, 本節冒頭でみた “Two Noriegas” の発見そのものです. 単語が $n+1$ 回現れる確率は, n 回現れる確率の $n/(n+1)$ 倍になっており, これから**単語は一度現れれば, バースト的に出現することになります.** この現象は頻度が同じでも α_v が小さい, すなわち一部の文書でのみ現れる単語 v ほど式(5.55)の近似が正確になり, より顕著になります. ポリア分布を使うことで, われわれは最初に述べた『シャーロック・ホームズの冒険』での単語のバースト性が成り立つことを, 数学的に説

明することができました.[*34]

5.4 潜在ディリクレ配分法 (LDA)

これまで, UM や DM では各文書が 1 つのトピックから生成されたと仮定してきましたが, 実際はこれには限界があると考えられます. たとえば, 「劇場の改修」を伝えるニュースでは, 演劇に関する言葉と建築に関する言葉を両方使うことになるでしょう. また, 数理経済学の論文では, 数学の話と経済学の話が混じっていると考えられます. これらを単一のトピックで表現しようとすると, すべての組み合わせに別々のトピックを用意しなければならなくなってしまいます. 組み合わせは 2 つとは限りませんから, これには厖大なトピック数が必要になります. また「数理経済学」と「開発経済学」のトピックが, 経済学という話題を共有していることもわからなくなってしまうでしょう.

よって, 各文書にトピックが 1 つだけあるのではなく, 図 5.20(a) のようにトピックの**確率分布** $\boldsymbol{\theta}$ があると考えた方がよいでしょう. たとえばトピックが $K=4$ 種類あり, それぞれ「演劇」「経済」「建築」「数学」だったとき, 上の劇場の改修のニュースは $\boldsymbol{\theta}=(0.7,0,0.3,0)$, 数理経済学の論文は $\boldsymbol{\theta}=(0,0.4,0,0.6)$ のようになると考えられます. 数学的には, 各文書 $\mathbf{d}$ についてトピック z の確率分布 $\boldsymbol{\theta}=\{p(z|\mathbf{d})\}$ を考えることになります. UM や DM は, これをあるトピック k についてだけ 1 をとるデルタ関数 $p(z|\mathbf{d})=\delta(z=k)$ で近似してしまうモデルとみなすことができます [*35].

この $\boldsymbol{\theta}$ は文書ごとに異なり, 最も簡単には 3 章で学習したディリクレ分布

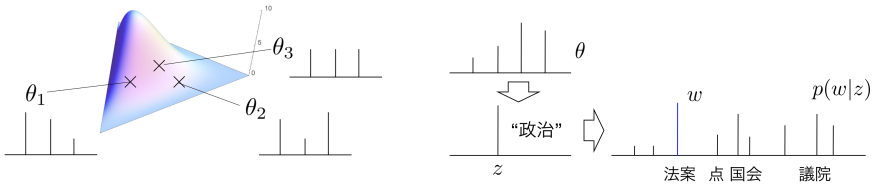
$$\boldsymbol{\theta} \sim \mathrm{Dir}(\boldsymbol{\alpha}) \quad (\boldsymbol{\alpha}=(\alpha_1,\cdots,\alpha_K)) \tag{5.57}$$

から生成されたと考えられます. 図 5.20(b) に示したように, この $\boldsymbol{\theta}$ から, 文書の各単語ごとに

---

*34 筆者が大学 (の文科系) に入学した 1993 年, 『数学セミナー』の誌上で岩波国語辞典の編者としても知られる国語学者の水谷静夫先生が, ある単語の頻度に Pólya-Eggenberger 分布が非常によく当てはまるが, 理由はわからない, と書いておられました[198]. それから 10 年以上が経ち, その理由がこうして, ポリア分布の生成モデルの形で数学的に説明できることがわかりました.

*35 なお, 単一トピックの UM や DM でも, 学習の際には $\delta(z=k)$ の推定値として図 5.9 のように $p(z|d)$ を考えることになりますが, 真の値はあくまで単一のトピックで, それを確率的に推定しているにすぎません. それに対して, ここでは真の値 $p(z|d)$ 自体が多項分布であり, その推定値として, 多項分布の分布であるディリクレ分布が事後分布として得られることになります.



(a) ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ から生成される各文書のトピック分布 $\boldsymbol{\theta}$.

(b) 1 単語ごとに $\boldsymbol{\theta}$ からトピック z を選び, z ごとのトピック–単語分布 $p(w|z)$ から単語 w が生成されます.

図 5.20: LDA の文書生成モデル. LDA では, 文書の各単語の位置ごとに異なる潜在トピックが存在し, そのトピックごとの確率分布から単語が生成されます.

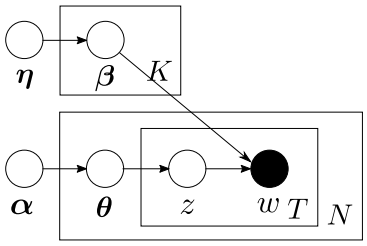
(1) まずトピック $z \sim \boldsymbol{\theta}$ をランダムに選び,
(2) z ごとのトピック–単語分布 $p(w|z)$ から単語 w を生成する

ことを繰り返して, 文書が生成されたとするモデルを考えることができます. このトピック–単語分布 $p(w|z)$ も, z ごとにディリクレ事前分布 $\mathrm{Dir}(\boldsymbol{\eta})$ から生成されたとしましょう. 以上の生成モデルを, アルゴリズムの形で図 5.21(a) に示しました. 学習の際には, 文書ごとに潜在変数 $\boldsymbol{\theta}$ のディリクレ事後分布を割り当てて推定するため, このモデルを**潜在ディリクレ配分法** (Latent Dirichlet Allocation, **LDA**) [185]といいます. LDA は後で説明する PLSI のベイズ化として, カリフォルニア大学バークレー校 (当時) の Blei らにより, 2001 年の論文[199]で最初に提案されました.

LDA のグラフィカルモデルを, 図 5.21(b) に示しました. 図 5.15 の UM のグラフィカルモデルと比べると, z が内側の箱の中に入っており, 文書に含まれる 1 つ 1 つの単語 w ごとに潜在トピック z があることを示しています. 100 万単語のコーパスがあれば, 100 万個の z があるわけです. よって LDA は, UM よりずっと複雑, かつ柔軟な確率モデルになっています. LDA では文書全体ではなく, 単語ごとに潜在トピックを考えるため, 以下では文書を単語列 $\mathbf{w} = w_1 w_2 \cdots w_T$ (実際には順番を考えないので単語集合) として説明します.

- For $k = 1, \cdots, K$,
  - Draw $\boldsymbol{\beta}_k \sim \mathrm{Dir}(\boldsymbol{\eta})$.
- For $i = 1, \cdots, N$,
  - Draw $\boldsymbol{\theta}_i \sim \mathrm{Dir}(\boldsymbol{\alpha})$.
  - For $t = 1, \cdots, T$,
    - Draw $z_{it} \sim \boldsymbol{\theta}_i$.
    - Draw $w_{it} \sim \boldsymbol{\beta}_{z_{it}}$.

(a) LDA の生成モデル



(b) LDA のグラフィカルモデル

図 5.21: LDA の生成モデルと, 対応するグラフィカルモデル. ●は観測された変数を, ○は潜在変数を表しています.

5.4.1 Gibbs サンプリングによる学習

LDA の学習法には (1) 最初に提案された変分ベイズ法[185], (2) 周辺化の考え方を取り入れた周辺化変分ベイズ法[200], (3) Gibbs サンプリング[201][202] などがあります [*36]. この中で, 5.2 節で説明した EM アルゴリズムのベイズ版である変分ベイズ法は数学的導出の難しさにもかかわらず, EM アルゴリズムの一種のため局所解の問題を逃れることができず, 図 5.22 に示したように, Gibbs サンプリング (MCMC 法) による学習が最終的に, 最も性能が高いことがわかっています. 導出や実装も容易なため, 本書では Gibbs サンプリングによる学習について説明します [*37]. 以下では 4.4.1 節の HMM のパラメータの学習について読んでいることを前提にしていますので, 難しいと感じる方は, 戻って復習してみてください.

上の生成モデルによれば, LDA で文書 $\mathbf{w} = w_1 w_2 \cdots w_T$ と背後にある潜在変数

(1) 各単語の潜在トピック $\mathbf{z} = z_1 z_2 \cdots z_T$,
(2) 文書の潜在トピック分布 $\boldsymbol{\theta}$,
(3) トピック–単語分布 $\boldsymbol{\beta} = \{\beta_{kv} = p(v|k)\}$

の同時確率は, 図 5.21(b) のグラフィカルモデルから

*36 期待値伝搬法 (EP) による学習も提案されていますが[203], この場合はすべての単語についてそのトピック事後分布を保持しなければならないため, 学習がスケールしないことが問題です.
*37 変分ベイズ法による学習について知りたい方は, [84]を参照してください.

$$p(\mathbf{w},\mathbf{z},\theta,\boldsymbol{\beta}|\boldsymbol{\alpha},\boldsymbol{\eta}) = p(\mathbf{w}|\mathbf{z},\boldsymbol{\beta})\,p(\mathbf{z}|\theta)\,p(\theta|\boldsymbol{\alpha})\,p(\boldsymbol{\beta}|\boldsymbol{\eta}) \tag{5.58}$$

と分解することができます. 具体的には, 217 ページで導入した指示関数 $\mathbb{I}()$ による記法を使うと,

$$\begin{aligned} &p(\mathbf{w}|\mathbf{z},\boldsymbol{\beta})\,p(\mathbf{z}|\boldsymbol{\theta})\,p(\boldsymbol{\theta}|\boldsymbol{\alpha})\,p(\boldsymbol{\beta}|\boldsymbol{\eta}) \\ &= \left(\prod_{t=1}^{T} p(w_t|z_t,\boldsymbol{\beta})p(z_t|\boldsymbol{\theta})\right)\cdot \mathrm{Dir}(\boldsymbol{\theta}|\boldsymbol{\alpha})\cdot\prod_{k=1}^{K}\mathrm{Dir}(\boldsymbol{\beta}_k|\boldsymbol{\eta}) \\ &\propto \underbrace{\prod_{t=1}^{T}\prod_{v=1}^{V}\prod_{k=1}^{K}\beta_{kv}^{\mathbb{I}(w_t=v)\mathbb{I}(z_t=k)}}_{\beta_{z_t w_t}}\cdot\underbrace{\prod_{t=1}^{T}\prod_{k=1}^{K}\theta_k^{\mathbb{I}(z_t=k)}}_{\theta_{z_t}}\cdot\prod_{k=1}^{K}\theta_k^{\alpha_k-1}\cdot\prod_{k=1}^{K}\prod_{v=1}^{V}\beta_{kv}^{\eta-1} \end{aligned} \tag{5.59}$$

(**LDA の文書同時確率**)

と表すことができます. 4 章 (217 ページ) で説明したように, パラメータ $\beta_{kv}=p(v|k)$ を表に出すために, $p(w_t|z_t,\boldsymbol{\beta})=\beta_{kv}$ (ただし $k=z_t$, $v=w_t$) を指示関数 $\mathbb{I}()$ を使って, 「ただし」の部分を数学的に

$$p(w_t|z_t,\boldsymbol{\beta}) = \prod_{v=1}^{V}\prod_{k=1}^{K}\beta_{kv}^{\mathbb{I}(z_t=k)\mathbb{I}(w_t=v)} \tag{5.60}$$

と表していることに注意してください. θ_k についても同様です.

実際には, 文書は $\mathbf{w}_1^N=\mathbf{w}_1,\mathbf{w}_2,\cdots,\mathbf{w}_N$ の N 個ありますから, データ全体の同時確率は, 対応する潜在トピックを $\mathbf{z}_1^N=\mathbf{z}_1,\mathbf{z}_2,\cdots,\mathbf{z}_N$, 各文書のトピック分布を $\boldsymbol{\theta}_1^N=\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\cdots,\boldsymbol{\theta}_N$ として

$$\begin{aligned} &p(\mathbf{w}_1^N,\mathbf{z}_1^N,\boldsymbol{\theta}_1^N,\boldsymbol{\beta}|\boldsymbol{\alpha},\boldsymbol{\eta}) \\ &\propto \prod_{i=1}^{N}\left[\prod_{t=1}^{T}\prod_{v=1}^{V}\prod_{k=1}^{K}\beta_{kv}^{\mathbb{I}(w_{it}=v)\mathbb{I}(z_{it}=k)}\cdot\prod_{t=1}^{T}\prod_{k=1}^{K}\theta_{ik}^{\mathbb{I}(z_{it}=k)}\prod_{k=1}^{K}\theta_{ik}^{\alpha_k-1}\right]\prod_{k=1}^{K}\prod_{v=1}^{V}\beta_{kv}^{\eta-1} \end{aligned} \tag{5.61}$$

と表されます. 記法を簡単にするために, 文書の長さ T を上では同じにしていますが, 実際は文書によって異なる値であることに注意してください.

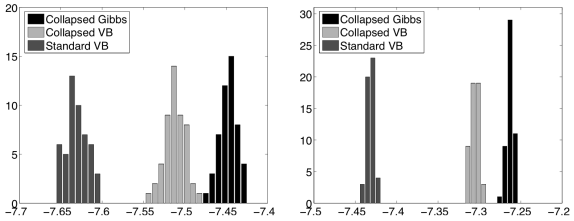


図 5.22: LDA の学習アルゴリズムの違いによる性能の比較 ([200]より引用). 左側は DailyKOS のニュース記事, 右側は NIPS の論文コーパスでの結果で, 横軸はテストデータの 1 単語あたりの対数尤度を表します. 異なる乱数で実験を複数回行っても, 変分ベイズ法に比べて, 黒棒の周辺化 Gibbs サンプリングがもっとも高い対数尤度を与えていることがわかります.

LDA の Gibbs サンプリング

式(5.61)のうち, わかっているのは観測された文書 $\mathbf{w}$ だけで, ほかはすべて未知の潜在変数ですので, $\mathbf{w}$ から推定します. LDA のパラメータ $\mathbf{z}, \boldsymbol{\theta}, \boldsymbol{\beta}$ は, 4 章で学んだ Gibbs サンプリングを使って, 順番にサンプリングすることを繰り返せば推定することができます.

$\mathbf{z}$ のサンプリング 文書 i の t 番目の単語 w_{it} のトピックを表す潜在変数 z_{it} は, 式(5.58)の 1 行目で z に関連する項は $p(\mathbf{w}|\mathbf{z}, \boldsymbol{\beta})p(\mathbf{z}|\theta)$ だけですから, 式(5.61)から対応する項を抜き出せば,

$$
\begin{aligned}
p(z_{it}|\mathbf{z}_{-it}, \boldsymbol{\theta}, \boldsymbol{\beta}) &\propto p(w_{it}|z_{it}, \boldsymbol{\beta})p(z_{it}|\mathbf{z}_{-it}, \boldsymbol{\theta}_i) \\
&= \beta_{kv} \cdot \theta_{ik} \quad (v = w_{it})
\end{aligned} \tag{5.62}
$$

となります. これは, 意味的には現在の文書を $\mathbf{w}_i = d$, 単語を $w_{it} = v$, サンプリングしたい $z_{it} = k$ とおくと,

$$
p(k|v, d) \propto p(v, k|d) = \underbrace{p(v|k)}_{\beta_{kv}} \underbrace{p(k|d)}_{\theta_{ik}} \tag{5.63}
$$

を計算している, ということです. したがって, z_{it} は確率

$$
p(z_{it} = k|\mathbf{z}_{-it}, \boldsymbol{\theta}, \boldsymbol{\beta}) = \frac{\beta_{kv}\theta_{ik}}{\sum_{k=1}^{K} \beta_{kv}\theta_{ik}} \tag{5.64}
$$

に従ってサンプリングすればよいことがわかります. この様子を, 図 5.23 に示しました.

$\boldsymbol{\theta}$ のサンプリング 同様に, 式(5.58)で θ について注目すると

$$p(\boldsymbol{\theta}\,|\,\mathbf{z},\boldsymbol{\alpha}) \propto p(\mathbf{z}\,|\,\boldsymbol{\theta})\,p(\boldsymbol{\theta}\,|\,\boldsymbol{\alpha}) \tag{5.65}$$

ですから, 式(5.61)から $\boldsymbol{\theta}$ に比例する項を抜き出して

$$p(\boldsymbol{\theta}_i\,|\,\mathbf{z}_i,\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} \theta_{ik}^{\alpha_k-1} \cdot \prod_{t=1}^{T}\prod_{k=1}^{K} \theta_{nk}^{\mathbb{I}(z_{it}=k)} = \prod_{k=1}^{K} \theta_{nk}^{\alpha_k-1+\sum_{t=1}^{T}\mathbb{I}(z_{it}=k)} \tag{5.66}$$

となり, $n(i,k)=\sum_{t=1}^{T}\mathbb{I}(z_{it}=k)$ とおけば, θ_i はディリクレ事後分布

$$p(\boldsymbol{\theta}_i\,|\,\mathbf{z}_i,\boldsymbol{\alpha}) = \mathrm{Dir}\,(\alpha_1+n(i,1),\cdots,\alpha_K+n(i,K)) \tag{5.67}$$

からサンプリングすればよいことがわかります.

$\boldsymbol{\beta}$ のサンプリング 最後に, 最も重要なトピック–単語分布 $\boldsymbol{\beta}$ のサンプリングを考えましょう. 式(5.58)で $\boldsymbol{\beta}$ に対応する項は

$$p(\boldsymbol{\beta}\,|\,\mathbf{w},\mathbf{z},\boldsymbol{\theta},\boldsymbol{\alpha}) \propto p(\mathbf{w}\,|\,\mathbf{z},\boldsymbol{\beta})p(\boldsymbol{\beta}) \tag{5.68}$$

ですから, $\boldsymbol{\beta}$ がトピック k ごとにディリクレ事前分布

$$p(\boldsymbol{\beta}_k) = \mathrm{Dir}\,(\eta,\cdots,\eta) \propto \prod_{v=1}^{V} \beta_{kv}^{\eta-1} \tag{5.69}$$

に従うとすれば, $\boldsymbol{\beta}_k$ の事後分布は式(5.61)から $\boldsymbol{\beta}$ に比例する項を抜き出して,

$$\begin{aligned} p(\boldsymbol{\beta}_k\,|\,\mathbf{w}_1^N,\mathbf{z}_1^N) &\propto \prod_{i=1}^{N}\prod_{t=1}^{T}\prod_{v=1}^{V} \beta_{kv}^{\mathbb{I}(w_{it}=v)\,\mathbb{I}(z_{it}=k)} \cdot \prod_{v=1}^{V} \beta_{kv}^{\eta-1} \\ &= \prod_{v=1}^{V} \beta_{kv}^{\eta-1+\sum_{i=1}^{N}\sum_{t=1}^{T}\mathbb{I}(w_{it}=v)\,\mathbb{I}(z_{it}=k)} \end{aligned} \tag{5.70}$$

となります. $m(k,v)=\eta+\sum_{i=1}^{N}\sum_{t=1}^{T}\mathbb{I}(w_{it}=v)\,\mathbb{I}(z_{it}=k)$ とおけば, こちらもディリクレ事後分布

$$p(\boldsymbol{\beta}_k\,|\,\mathbf{w}_1^N,\mathbf{z}_1^N) = \mathrm{Dir}\,(\eta+m(k,1),\cdots,\eta+m(k,V)) \tag{5.71}$$

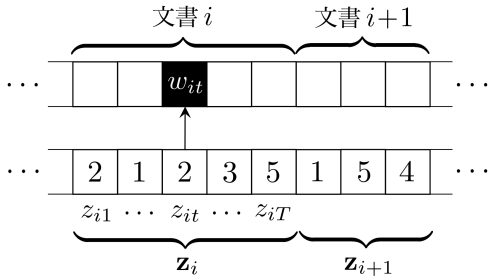


図 5.23: LDA の Gibbs サンプリングの様子. 各単語 w_{it} を生成した潜在トピック z_{it} を, 事後分布からランダムにサンプリングして更新することを繰り返します.

からサンプリングすることができます.

式(5.64), 式(5.67), 式(5.71) に従って $\mathbf{z}, \boldsymbol{\theta}, \boldsymbol{\beta}$ を次々とサンプリングすることを繰り返せば, LDA のパラメータを求めることができます. 実は, データ量やトピック数は圧倒的に少ないものの, 本質的に同じモデルが LDA より 1 年前にバイオインフォマティクスの分野で Pritchard ら[204]によって提案されており, 推論にはこの方法が使われていました.

5.4.2 周辺化 Gibbs サンプリングによる学習

しかし, パラメータ数が桁違いに大きい自然言語の場合は, HMM (223 ページ) や UM (266 ページ) でもわれわれが行ったように, パラメータについて期待値をとり, 可能な限り積分消去することで, より効率的なサンプリングを行うことができます. 実際には, こちらの方法で学習を行うのがよいでしょう.

式(5.62)で z_{it} のサンプリングを行う際に, θ_{ik} が必要になりました. ただし, z_{it} 以外の $\mathbf{z}_i \backslash z_{it}$ はわかっているのですから, $\boldsymbol{\theta}_i$ の事後分布は式(5.67)から, $\mathbf{z}_i \backslash z_{it}$ の中でトピック k に割り当てられた単語の数を $n'(i,k)$ とおくと,

$$p(\boldsymbol{\theta}_i \,|\, \mathbf{z}_i \backslash z_{it}) = \mathrm{Dir}\,(\alpha_1 + n'(i,1), \cdots, \alpha_K + n'(i,K)) \tag{5.72}$$

です. よって, その期待値

$$\mathbb{E}[\theta_{ik} \,|\, \mathbf{z}_i \backslash z_{it}] = \frac{\alpha_k + n'(i,k)}{\sum_k (\alpha_k + n'(i,k))} \tag{5.73}$$

アルゴリズム 7: LDA の周辺化 Gibbs サンプリング

```
1: for i in 1,⋯,N do                    (* 初期化 *)
2:   for t in 1,⋯,T do
3:     z_it = randint(K)                (* 1⋯K の乱数で初期化 *)
4:     n'(i,z_it)++
5:     m'(w_it,z_it)++
6: while 収束するまで do                 (* Gibbs サンプリング *)
7:   for i in randperm(N) do            (* 文書 i をランダムに選ぶ *)
8:     for t in 1,⋯,T do
9:       n'(i,z_it)--                   (* カウントを減らす *)
10:      m'(w_it,z_it)--
11:      z_it を式(5.76)に従ってサンプリング
12:      n'(i,z_it)++                   (* カウントを増やす *)
13:      m'(w_it,z_it)++
14: 式(5.77)から θ_1^N, β の期待値を求める
```

図 5.24: 周辺化 Gibbs サンプリングによる LDA の学習アルゴリズム.

を使えばよいでしょう.

同様にして β_{kv} についても, その期待値で置き換えることができます. $\boldsymbol{\beta}_k$ の事後分布は式(5.71)から,

$$p(\boldsymbol{\beta}_k \,|\, \mathbf{w}_1^N, \mathbf{z}_1^N \backslash z_{it}) = \mathrm{Dir}\,(\eta + m'(k,1), \cdots, \eta + c'(k,V)) \tag{5.74}$$

となることに注意しましょう. ここで $m'(k,v)$ は z_{it} を除いたすべての潜在変数 $\mathbf{z}_1^N \backslash z_{it}$ の中で, 単語 v にトピック k が割り当てられた回数を表しています. よって, 式(5.64)の β_{kv} の部分は

$$\mathbb{E}[\beta_{kv} \,|\, \mathbf{z}_1^N \backslash z_{it}] = \frac{\eta + m'(k,v)}{\sum_{v=1}^{V} (\eta + m'(k,v))} \tag{5.75}$$

となります.

以上より, 式(5.75)と式(5.73)から, z_{it} のサンプリングは式(5.64)にかえて, 次の式で行うことができます. [*38]

*38 実装の際には, 式(5.76)の第 1 項の分母の $\Sigma_{k=1}^{K}(\alpha_k + n'(i,k))$ は k に依存しませんので, 比例式としては不要です. また, 第 2 項の分母の $\Sigma_{v=1}^{V}(\eta + m'(k,v))$ も実際に V 個の和を毎回計

$$(5.76)\quad p(z_{it}=k\,|\,w_{it}=v,\mathbf{z}_i\backslash z_{it}) \propto \frac{\alpha_k+n'(i,k)}{\sum_{k=1}^{K}(\alpha_k+n'(i,k))}\cdot\frac{\eta+m'(k,v)}{\sum_{v=1}^{V}(\eta+m'(k,v))}$$

(LDA の周辺化 Gibbs サンプリングの公式)

この場合, $\boldsymbol{\theta}$ や $\boldsymbol{\beta}$ のサンプリングは不要で, $\mathbf{z}$ だけをサンプルすれば学習できることに注意してください. なお, 式(5.76) (および式(5.64)) による Gibbs サンプリングは, コーパス上のすべての単語についてトピック $1,\cdots,K$ に属する確率を計算するため, K が増えると計算量が非常に大きくなります. しかし, 驚くべきことに, 巧妙な方法で K にかかわらず $O(1)$ の計算量で式(5.76)によるサンプリングは実行でき[205], K が 1000 から 10000 を超えるような大規模なトピックモデルの計算パッケージ LightLDA [206]も公開されています [*39]. $\mathbf{z}$ のサンプリングが収束すれば, $\boldsymbol{\theta}$ と $\boldsymbol{\beta}$ は式(5.67)および式(5.71)の期待値をとることで,

(5.77)

$$\mathbb{E}[\theta_{nk}\,|\,\mathbf{z}_i]=\frac{\alpha_k+n(i,k)}{\sum_{k=1}^{K}(\alpha_k+n(i,k))},\quad \mathbb{E}[\boldsymbol{\beta}_{kv}\,|\,\mathbf{z}_1^N]=\frac{\eta+m(k,v)}{\sum_{v=1}^{V}(\eta+m(k,v))}$$

と求めることができます. この学習アルゴリズムを, 図 5.24 に示しました.

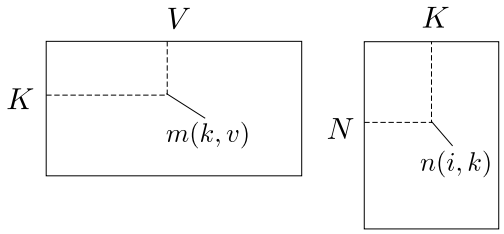


図 5.25: LDA の周辺化 Gibbs サンプリングで保持する統計量 $m(k,v)$, $n(i,k)$ の行列. 行列の各行の頻度を生成した多項分布がそれぞれディリクレ分布に従っており, 全体の確率は, ポリア分布の積になります.

算する必要はなく, $m'(k)=\Sigma_{v=1}^{V}m'(k,v)$ を保持しておいて更新すれば, $V\eta+m'(k)$ で求められることに注意してください.

*39 LightLDA はデータに特殊なフォーマットを必要とするため, 本書と同じ SVMlight 形式のデータで簡単に使うためのラッパー `lightlda.sh` を, 筆者が `http://chasen.org/~daiti-m/dist/lightlda.sh/` で公開しています.

表 5.9: LDA ($K=50$) で教師なし学習された livedoor コーパスのトピックの例. 表 5.1 の教師ラベル (カテゴリ) では表現できない, 細かい意味的なトピックが学習されていることがわかります.

トピック 1		トピック 7		トピック 43		トピック 48	
仕事	0.0357	応募	0.0435	料理	0.0276	結婚	0.0464
転職	0.0297	名	0.0431	食べ	0.0210	女	0.0389
会社	0.0246	プレゼント	0.0370	店	0.0120	女性	0.0367
情報	0.0225	様	0.0338	味	0.0118	彼	0.0334
年収	0.0179	当選	0.0291	野菜	0.0105	男性	0.0303
人	0.0161	月	0.0226	食べる	0.0086	男	0.0258
自分	0.0160	キャンペーン	0.0224	食	0.0080	恋愛	0.0244
万	0.0160	日	0.0193	酒	0.0074	相手	0.0216
livedoor	0.0158	終了	0.0177	ビール	0.0071	彼女	0.0171
求人	0.0131	場合	0.0172	食事	0.0070	自分	0.0140

LDA の周辺化尤度の計算 なお, この場合は周辺化 Gibbs サンプリングの目的関数は, 式(5.61) において $\boldsymbol{\theta}_1^N$ と $\boldsymbol{\beta}$ を周辺化した

$$p(\mathbf{w}_1^N, \mathbf{z}_1^N \,|\, \boldsymbol{\alpha}, \boldsymbol{\eta}) = p(\mathbf{w}_1^N \,|\, \mathbf{z}_1^N, \boldsymbol{\eta})\, p(\mathbf{z}_1^N \,|\, \boldsymbol{\alpha}) \tag{5.78}$$

になります[202]. この各項は, 図 5.25 に示したように $n(i,k)$ および $m(k,v)$ を行列で表せば, 各行にそれぞれ潜在的な多項分布 $\boldsymbol{\theta}_i$, $\boldsymbol{\beta}_k$ が存在してディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$, $\mathrm{Dir}(\boldsymbol{\eta})$ に従い, これから各行の頻度 $n(i,:)$ および $m(k,:)$ が生まれたことを意味します. したがって, この尤度は, 3 章の式(3.55)で学習したポリア分布になり, 式(5.78)の右辺の各項は $m(k) = \sum_v m(k,v)$ とおくと,

$$\begin{cases} p(\mathbf{z}_1^N \,|\, \boldsymbol{\alpha}) & = \displaystyle\prod_{i=1}^{N} \left[\frac{\Gamma(\sum_k \alpha_k)}{\Gamma(T_i + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n(i,k))}{\Gamma(\alpha_k)} \right] \\ p(\mathbf{w}_1^N \,|\, \mathbf{z}_1^N, \boldsymbol{\eta}) & = \displaystyle\prod_{k=1}^{K} \left[\frac{\Gamma(V\eta)}{\Gamma(V\eta + m(k))} \prod_{v=1}^{V} \frac{\Gamma(\eta + m(k,v))}{\Gamma(\eta)} \right] \end{cases} \tag{5.79}$$

で計算されることに注意してください. これから式(2.69)を使って, 学習データの 1 単語あたりのパープレキシティを求めることができます.

LDA の実験

LDA は単語ごとに潜在トピックが存在する確率モデルのため, Python のみで実装すると, 非常に計算が遅くなります. モジュールを C 言語にコンパイルして高速化する Cython を使った実装を筆者が公開していますので, 公開ページ [*40], または本章のサポートページから `lda.py-0.2.tar.gz` をダウンロードした後,

```
% tar xvfz lda.py-0.2.tar.gz
% cd lda.py-0.2/
% make
```

でモジュールをコンパイルし, 準備することができます. これには Cython が必要ですので, 事前に `% pip install cython` などを実行してインストールしておいてください.

この上で, livedoor コーパスについて

```
   % lda.py -K 50 -N 400 livedoor.dat model livedoor.lex
⇒  LDA: K = 50, iters = 400, alpha = 1, beta = 0.01
   loading data.. documents = 7367, lexicon = 16946, nwords = 1838414
   initializing..
   Gibbs iteration [400/400] PPL = 4761.2851
   saving model to model ..
   including dictionary.
   done.
```

を実行すると, トピック数 $K=50$ の LDA を学習することができます. この結果の一部を, 表 5.9 に示しました. 執筆時の環境では, この計算には約 6 分かかります. 上では MCMC の繰り返し数を 400 にしていますが, `model.log` に保存される学習データのパープレキシティの様子を見て, 収束するまで適宜増やすといいでしょう. 表 5.9 からわかるように, LDA により, 文書全体を 1 トピックとする UM や教師ラベルでは表現できない, 仕事, プレゼント, 食事, 恋愛などの細かいトピックが自動的に学習されていることがわかります.

表 5.10(a) に, 同様にして `jawiki.txt` について表 5.7 の UM と同じ $K=100$ トピックの LDA を学習した場合のトピックの一部を示しました. LDA では, 文書全体を 1 トピックで表現する必要はないため, トピック 30 のように「的」「化」

---

*40 `http://chasen.org/~daiti-m/dist/lda-python/`

表 5.10: LDA ($K=100$) で学習した日本語 Wikipedia コーパスのトピックの例.

(a) 生成確率 $p(w|z)$ を使った場合

トピック 10		トピック 20		トピック 30		トピック 100	
県	0.2955	世紀	0.0832	的	0.2476	月	0.3766
市	0.2018	時代	0.0777	化	0.0267	日	0.3252
福岡	0.0246	歴史	0.0341	基本	0.0255	年	0.2580
埼玉	0.0239	初期	0.0291	人間	0.0243	テン	0.0030
広島	0.0229	頃	0.0286	方法	0.0243	ルー	0.0017
兵庫	0.0229	年代	0.0224	主	0.0233	支え	0.0013
愛知	0.0215	説	0.0217	一般	0.0223	姓	0.0011
千葉	0.0194	当時	0.0215	表現	0.0206	セントラル	0.0011
九州	0.0188	記	0.0172	用い	0.0200	政治	0.0008
名古屋	0.0172	古代	0.0167	手法	0.0176	会議	0.0008

(b) 正規化自己相互情報量 NPMI (w,z) を使った場合

トピック 10		トピック 20		トピック 30		トピック 100	
県	0.7763	世紀	0.6493	的	0.7374	月	0.7545
市	0.6981	時代	0.5917	基本	0.5516	日	0.7281
福岡	0.5526	頃	0.5474	方法	0.5420	年	0.5677
埼玉	0.5511	初期	0.5386	人間	0.5392	テン	0.4269
広島	0.5488	年代	0.5383	手法	0.5311	ルー	0.3910
兵庫	0.5488	歴史	0.5379	主	0.5289	支え	0.3356
愛知	0.5442	説	0.5359	表現	0.5238	モー	0.3236
千葉	0.5350	記	0.5305	分析	0.5201	セントラル	0.3170
名古屋	0.5306	中世	0.5260	図	0.4977	ウイング	0.2552
新潟	0.5306	不明	0.5195	成分	0.4944	カルロス	0.2548

といった機能語に対応するトピックができることが多く [*41], 生成確率 $p(w|z)$ を使っても, 他のトピックで機能語が上位に来ることはあまりありません. 表 5.10(b) に示したように, この場合も NPMI を用いることで, 各トピックに相関する単語をより明確に取り出すことができます. 図 5.26 に, 各文書 $\mathbf{d}$ に対する潜在トピック分布 $\theta=\{p(z|\mathbf{d})\}$ を示しました. 同じトピック数の図 5.9 の UM の場合と比べると, $p(z|\mathbf{d})$ がどれか 1 つのトピックに集中することがなく, 各文書が複数の潜在トピックからなる様子が学習されているのがわかります.

*41 293 ページの方法でハイパーパラメータ α_k も推定して学習すると, こうした結果になりやすいことが確かめられています[207].

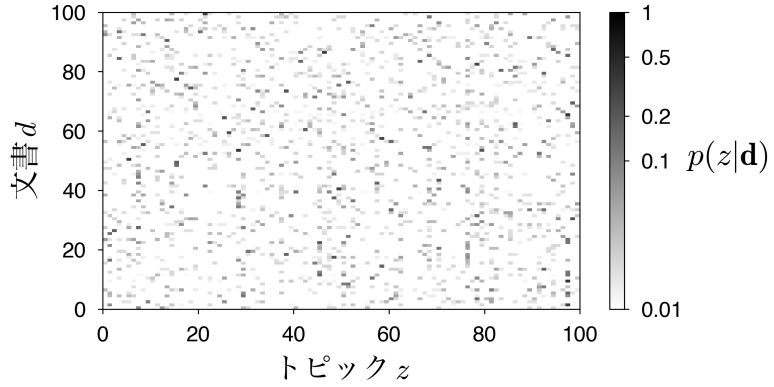


図 5.26: 日本語 Wikipedia コーパス `jawiki.txt` から学習された LDA による, 各文書のトピックへの所属確率 $p(z|\mathbf{d})$ (最初の 100 文書). トピック数は $K=100$ としました. 図の各行がそれぞれ文書に対応しており, 複数の潜在トピックを確率的に持っていることがわかります. 図 5.9 の UM の場合と比較してみましょう.

5.4.3 LDA の幾何的解釈

LDA は, 幾何的には何をしていることに相当するのでしょうか. LDA では, トピック分布 $\boldsymbol{\theta}$ が決まると, 文書の各単語は

1. トピック $z \sim \boldsymbol{\theta}$ をサンプルし,
2. 単語 $w \sim p(w|z)$ をサンプルする

という 2 段階で生成されます. 確率で書くと, $\boldsymbol{\theta}$ から w と z を生成する確率は

$$p(w, z|\boldsymbol{\theta}) = p(w|z)p(z|\boldsymbol{\theta}) \tag{5.80}$$

となっているわけです.

しかし, よく考えるとわれわれが観測しているのは w だけなのですから, z については和をとって周辺化してしまってもよいでしょう.[*42] つまり, w は確率分布

$$p(w|\boldsymbol{\theta}) = \sum_z p(w, z|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(w|z=k)p(z=k|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(w|k)\,\theta_k \tag{5.81}$$

[*42] これを 229 ページの脚注のように, Rao-Blackwell 化ともいうのでした.

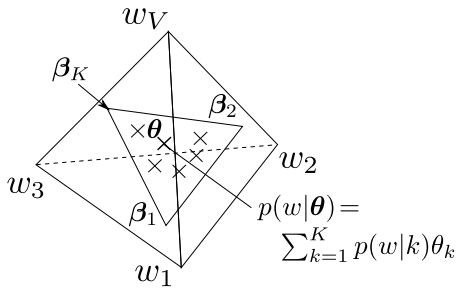


図 5.27: LDA の幾何的表現. LDA は×で示した単語の生成確率をモデル化できるよう, 外側の V 次元の単語単体の中で内側の K 次元のトピック単体 $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ の位置を最適化し, 次元圧縮を行う統計モデルです.

に従う 1 段階で生成されたと考えても, 数学的には等価になります.

トピック分布 $p(w|k)$ は, k ごとの単語の確率分布 $\boldsymbol{\beta}_k = (p(w_1|k), \cdots, p(w_V|k))$ のことですから, これは図 5.14 でみたように, 単語単体の中の 1 点になります. 図 5.27 に示したように, 式(5.81)は, テキストが $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ を $\theta_1, \cdots, \theta_K$ の割合で内分した点 $p(w|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(w|k)\theta_k$ から生成されたことを表しています. このように, V 次元の単語単体の中にあって $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ によって張られる K 次元の単体のことを, **トピック単体**といいます.

$p(w|\boldsymbol{\theta})$ はこのトピック単体の中にあるのですが, 一般に $K \ll V$ ですから, トピック単体 $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ をうまく選ばないと, 文書を生成した確率分布 $\mathbf{p}$ を表現することができません. このように,

(1) トピック, すなわちトピック単体の「角」$\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$ と,
(2) 文書ごとの $\boldsymbol{\theta}$, すなわち上で定まるトピック単体内の各文書の位置

を同時に最適化しているのが LDA です. このとき $\boldsymbol{\theta}$ にはディリクレ事前分布を仮定して, 各文書にディリクレ事後分布を割り当てる (allocation) ため, **潜在ディリクレ配分法**とよばれているわけです.

空間内に, データをうまく表現する低次元の部分空間を張るという意味では, これは実数値 (ユークリッド空間) における主成分分析 (PCA) に似ています. ただし PCA と異なり, LDA は離散データのために単体上で定義された統計モデルなのが特徴で, 低頻度のデータでもうまく扱うことができます. Buntine ら [208]は, LDA およびその拡張を統一して離散 PCA (Discrete PCA) とよんで

います.

メモ：ディリクレ分布の $\boldsymbol{\alpha}$ のサンプリング

LDA のハイパーパラメータは, トピック分布 $\boldsymbol{\theta}$ およびトピック–単語分布 $\boldsymbol{\beta}$ を生成するディリクレ分布のハイパーパラメータ $\boldsymbol{\alpha}, \eta$ ですが, これらはどうやって決めたらいいのでしょうか.

一般的にはトピック数を K として, $\boldsymbol{\alpha} = (50/K, \cdots, 50/K)$, $\eta = 0.01$ のように発見的に決められることが多いのですが, これらのハイパーパラメータの影響を詳しく調べた研究[209]によると, これらもデータから学習した方が, よいモデルとなることが示されています. とくに, 非対称な $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_K)$ を推定することで, α_k が大きいトピックに「が」「の」, 英語なら "of", "the" といった機能語 [*43] が集まり, 他のトピックにこれらが混ざることが少なくなります. また, トピック数を増やしても不要なトピックは α_k が小さく, 自然に使われなくなることも報告されています.

$\boldsymbol{\alpha}$ と η についての尤度関数は式(5.79)ですから, 数学的にはこれは, このポリア分布のハイパーパラメータを推定することと等価です. ただし, このカウント $n(i,k)$, $m(k,v)$ はあくまで学習中の $\mathbf{z}_1^N$ から計算される値ですから, 学習中に式(3.57)を使って $\boldsymbol{\alpha}$ を最適化してしまうと, 局所解に陥る危険があります.[*44] よって, きちんと $\boldsymbol{\alpha}$ のベイズ推定を行うのが正しい方法でしょう.

ポリア分布を表す式

$$p(\mathbf{n}|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N + \sum_k \alpha_k)} \prod_{k=1}^{K} \frac{\Gamma(\alpha_k + n_k)}{\Gamma(\alpha_k)} \tag{5.82}$$

は, α_k がガンマ関数 $\Gamma()$ の中に含まれているため, そのままではサンプリングできる形になりません. しかし, 巧妙な補助変数 θ, x を導入すると, α_k はガンマ事後分布からサンプリングすることができます. 導出はやや複雑なため, 詳しくは付録 B を参照してください.

---

*43 もしこうした機能語を「ストップワード」として除いても, 257 ページで説明したように, 同様に意味の薄い語が必ず出現し, 機能語とそれ以外は簡単に二分することができません.

*44 すなわち, 学習途中で $\boldsymbol{\alpha}$ を最適化してしまうと, 学習アルゴリズム全体が Gibbs サンプリ

5.4.4 トピックモデルの評価と拡張

こうして学習したトピックモデルが, 文書の「よいモデル」になっているかどうかは, どうやって測ればよいのでしょうか. LDA は先に説明したように, 単語単体上の密度推定を行う統計モデルですから, テストデータに高い確率を与えられるかが, 最も素直な評価方法といえるでしょう. LDA の評価方法について詳しく調べた研究[207]では, これにはテストデータの各文書について, 前半からトピック分布を学習し, それに基づいて後半の単語の予測パープレキシティを計算する方法が最もよかったことが報告されています.

サポートサイトの本章のフォルダに, この評価スクリプト `ldaeval.py` を示しました. LDA は順番を考慮しない単語集合のモデルですから, 評価の際には各文書で単語をランダムにシャッフルし, 前半 $\mathbf{w}=w_1\cdots w_{T/2-1}$ から計算したトピック分布 $\boldsymbol{\theta}$ を用いて, 後半 $\mathbf{w}'=w_{T/2}\cdots w_T$ の確率を

$$p(\mathbf{w}'|\boldsymbol{\theta})=\prod_{t=T/2}^{T}p(w_t|\boldsymbol{\theta})=\prod_{t=T/2}^{T}\sum_{k=1}^{K}p(w_t|z_t=k)\underbrace{p(z_t=k|\boldsymbol{\theta})}_{=\theta_k} \tag{5.83}$$

のように計算します. なお, $\mathbf{w}$ からそれを生成した $\boldsymbol{\theta}$ を求めるには, 式(5.76)の Gibbs サンプリングを行うこともできますが, パラメータはすでに学習済みですので, 変分ベイズ EM アルゴリズムを用いると高速に行えます. 詳しくは, `ldaeval.py` のスクリプトの中身および[84]を参照してください.

図 5.28 に, livedoor コーパスのうちランダムな 367 文書をテストデータ, 残りの 7,000 文書を学習データとした場合の予測パープレキシティを, 各トピック数 K について示しました. LDA はベイズ的な手法のため, K を大きくしても過学習して性能が落ちることはなく, パープレキシティはゆっくり減少していくことがわかります. また, 付録 BE の方法でハイパーパラメータ $\boldsymbol{\alpha},\boldsymbol{\eta}$ を推定した場合, 固定の場合と比べて予測精度が上がっていることも読みとれます.

トピックの直接評価

上ではトピックモデルの性能を予測精度で評価しましたが, これは得られたトピックが意味的まとまりの高い, 「よいトピック」であることを保証している

ングではなく, 267 ページのモンテカルロ EM アルゴリズムを行うことになってしまいます.

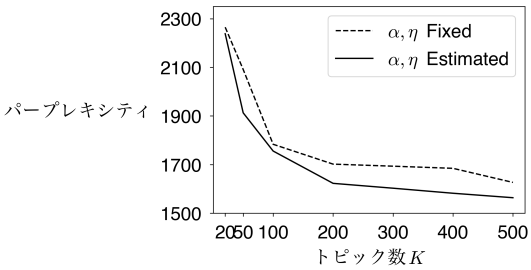


図 5.28: livedoor コーパスにおけるテストデータの予測パープレキシティとトピック数 K の関係. ハイパーパラメータ $\boldsymbol{\alpha}, \boldsymbol{\eta}$ を固定した場合 (Fixed) に対して, それらもベイズ推定した場合 (Estimated) の方が, 一貫して低いパープレキシティを達成することがわかります.

わけではありません. 「よいトピック」であることは, どうやって測ればいいのでしょうか? トピックモデルは教師なし学習ですので, この問題にはもちろん「正解」はないのですが, いくつかの方法が提案されています[210, 211]. 以下では, 各トピック k について, 生起確率 $p(w|k)$ の高い順に N 語 (たとえば $N = 10$) とった単語集合を「トピック語」$t(k)$ とよぶことにします.

Word Intrusion 各トピックが意味的によくまとまっていれば, トピック語に無関係な語 (侵略者, intruder) を混ぜてもたやすく検出できるはずです. 逆にノイズが多いトピックなら, どれが侵略者なのかを判断できないでしょう. たとえば図 5.29 で, A のトピック語に混ぜた “京都” は簡単に発見できますが, B に混ぜた “大阪” はすぐには見分けられません. この判断は人間の被験者に行ってもらうこともできますし, サポートベクトル回帰 (SVR) で自動化する方法も提案されています[211]. *45

Topic Coherence トピックが意味的にまとまっているかを, 直接測ることもできます. 5.2.1 節で議論したように, NPMI を使うとトピックと相関の高い単語を取り出すことができますから, よいトピックであれば, それらの単語どうしの NPMI も高くなるはずです. トピック語のすべてのペア (v, w) について, 外部の大きなテキストでの共起から計算した NPMI(v, w) の平均値を Coherence

*45 `https://github.com/jhlau/topic_interpretability` で, 自動評価のための評価スクリプトが公開されています.

トピック A	酸, 酵素, **京都**, 化合, 合成, 水素, 有機, 触媒
トピック B	湖, 不足, キル, PK, 通勤, **大阪**, テレビジョン

図 5.29: Word Intrusion の例. 意味的まとまりの高いトピック A では, 太字で示した侵略者 (intruder) はすぐに見分けることができますが, 意味的まとまりの低いトピック B では, どれが侵略者なのか判断がつきません.

と定義すると, これは上の Word Intrusion および人間の判断と高い相関を持つことが示されています[211].

Topic Uniqueness トピックが意味的にまとまっていても, ほとんど同じトピックが複数あるのは望ましくないでしょう. 各トピックのトピック語を「文書」とみなしたとき, ある単語 w が複数のトピック語に現れていれば, 309 ページで学ぶ文書頻度 (=何個のトピック語に現れたか) $\mathrm{df}(w)$ は 1 より大きくなります. よって, その逆数をとって平均し,

$$\mathrm{TU} = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{N}\sum_{w\in t(k)}\frac{1}{\mathrm{df}(w)} \tag{5.84}$$

を計算すれば, トピック語がすべて異なっていれば 1, 重複があると <1 になるはずです. これを Uniqueness と定義して評価する方法が提案されています[212].

ただし, 式(5.84)の指標はトピック数 K の違いについてロバストではありません. K が大きくなれば, ある単語が他のトピック語にも偶然含まれる可能性は, それだけ大きくなるからです. 式(5.84)は本質的に, すべてのトピック語をまとめた集合での各単語 w の出現回数 $\mathrm{df}(w)$ を計算しているだけですから, それをユニグラム分布 $p(w) = \mathrm{df}(w)/(NK)$ に直せば, 2 章で学習した $p(\cdot)$ のエントロピー (式(2.66)) $H(p(\cdot))$ を指標とすればよいでしょう. この最大値は, すべての単語が 1 回ずつ現れる (=トピック語に重複がない) 場合で, $\log(NK)$ になります. よって最大値との比をとり,

$$\mathrm{TU{+}{+}} = H(p(\cdot))/\log(NK) \tag{5.85}$$

を指標とすれば, これは K に依存しません. この指標 (本書のオリジナルです) を TU++とよぶことにしましょう.

こうした評価は, LDA に限らずトピックモデル全般に用いることができます.

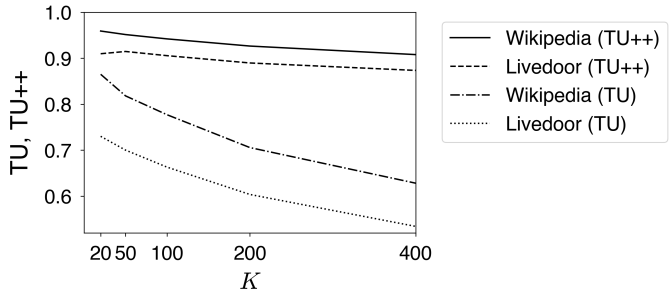


図 5.30: livedoor コーパスおよび日本語 Wikipedia で計算した TU,TU++の値とトピック数 K. K の選択には使えませんが, 同じトピック数での比較のためには有効です.

5.2 節の UM について, EM アルゴリズムで推定した表 5.6 のトピックと, Gibbs サンプリングで計算した表 5.7 のトピックを比べてみましょう. サポートサイトの `coherence.py` を使って計算すると,

```
   % coherence.py model/um.jawiki.K100 data/ja.text8.txt
⇒ obtaining topic words..
   computing cooccurrences..
   calculating coherence..
   average = -0.0997
```

のようになります. ここでは, 日本語 text8 を共起計算のためのコーパスとして用いました.[*46] EM アルゴリズムの場合は上のように −0.0997, Gibbs サンプリングの場合は −0.0684 となり, 確かに Gibbs サンプリングの方が意味的まとまりの高いトピックが学習されていることが定量化されました.

また Uniqueness については, livedoor コーパスについて `uniqueness++.py` で TU および TU++をさまざな K について計算すると図 5.30 のようになり, $K = 50$ 程度でやや TU++の値が高くなることがわかります. ただし, Wikipedia の場合は指標は K とともに一様に減少するようです.

トピック数 K の選択 トピックモデルにおいて, トピック数 K の選択は重要な問題です. 上でみたように, 予測パープレキシティ, Uniqueness (TU++) のい

---

*46 この 10 倍のサイズがある 145 ページの `ja.text9` を使っても, 統計値はほとんど同じになりました.

ずれも, それだけでトピック数を適切に決めることはできません.[*47] 理論的には, LDA のトピック数は階層ディリクレ過程 (HDP) を用いた HDP-LDA で推定でき, この問題は解決されています[85]. ただし, HDP の理解と実装には測度論の知識が必要となり, 本書の範囲を超えますので, 簡便には (1) HDP-LDA が実装されている gensim などのパッケージ [*48] を用いるか, または (2) 4 章で行ったように K を大きめにとって $\alpha<1$ と小さく設定するか, あるいは (3) $\boldsymbol{\alpha}$ をサンプリングして学習することで, 実質的に必要なトピックだけを残すようにするとよいでしょう. HDP による無限モデルに興味のある方は, 優れた導入である[19]を参照してください.

トピックモデルの拡張

2001 年に提案された LDA はさまざまな分野に適用され, 拡張されています. LDA は完全に教師なしでコーパス中の潜在トピックと各文書のトピック分布を学習するものですが, 解釈を容易にするため, 実用的には一部のトピックに「種」となるキーワードを与えられると便利です. 最近これを行う, keyATM [213]とよばれるモデル [*49] が政治学方法論の分野で提案されました. 政治学では, トピック–単語分布をロジスティック回帰でモデル化するテキストのスパース加法モデル[214]をベースにして文書の共変量をモデルに含めることのできる, 構造化トピックモデル (STM) [215]がよく使われています [*50]. LDA は単語を離散的にとらえるため, 3 章で学習した単語ベクトルを使用することで意味の類似性をより正確にとらえる, 埋め込みトピックモデル[216] も最近提案されています. 研究レベルですが, 筆者は潜在トピックに未知の階層構造があると考え,

---

∗47 LDA で学習データの確率が最も高くなるのは, トピックの数と単語数 V が等しく, 各トピックで β_{kv} がある単語だけ確率 1, 他は 0 となる場合です. しかし, 式(5.77)の $\boldsymbol{\beta}$ の事後分布から, $\eta>0$ のときにそうなる確率は 0 です. したがって η を固定すれば, 式(5.79)のエビデンス $p(\mathbf{w}_1^N \mid \mathbf{z}_1^N, \boldsymbol{\eta})$ を最大にするトピック数 K を求めることができます[202]. しかしこれは η に依存し, η=0.1 と η=0.01 ではまったく違うトピック数になってしまいます. 付録 B の式(B.13)から η をサンプリングする場合, K が大きいほど推定される η は小さくなり, この方法で最適な K を選ぶことはできません.

∗48 `https://radimrehurek.com/gensim/models/hdpmodel.html`

∗49 `https://keyatm.github.io/keyATM/`で R による実装が公開されています.

∗50 `https://www.structuraltopicmodel.com/`および `https://github.com/bstewart/stm` で R による実装が公開されています.

テキストだけから無限の深さと無限個の分岐の可能性をもつトピック木構造をMCMC 法で学習する, 無限階層トピックモデル ihLDA を提案しています[217].

5.1 節のように文書にラベル y がある場合, 潜在トピック分布 $\boldsymbol{\theta}$ からの [*51] ロジスティック回帰で y が生成されたと考えてモデル全体を学習するのが教師ありトピックモデル [218]で, これにより, ラベルの情報も有効に活用して潜在トピックを求めることができます. こうした拡張については, 筆者の統計数理研究所でのトピックモデル公開講座の資料 [*52] や, 佐藤によるトピックモデルの専門書[84]などを参照してください.

5.5　ニューラル文書モデルと独立成分分析

LDA は, 各文書にディリクレ分布に従う潜在的なトピック分布 $\boldsymbol{\theta}$ があると考え, $\boldsymbol{\theta}$ とトピックの両方をデータから学習できる, 解釈性に優れたモデルです. しかし, LDA にもいくつかの問題があることに注意が必要です.

- 一つは, 文書の内容を表現する $\boldsymbol{\theta}$ が多項分布に制限されていることです ($\sum_k \theta_k = 1, \theta_k \geq 0$). こうすると, たとえば経済と数学の両方の内容を持つ文書は, 経済トピックが強いほど数学トピックは弱くなり, その逆も成り立ちます. また, あまり内容がなかったり短い文書でも, $\boldsymbol{\theta}$ は内容の濃いテキストと同様に, 和が 1 の確率分布になってしまいます.
- 二つ目は, 単語を離散的にとらえているため, 3 章で学習した単語ベクトルのように単語間の関係を精密にモデル化できないことです. たとえば, “取る” と “取得” のようにほぼ同じ意味の言葉でも, LDA では「同じ潜在トピックから生成される確率が高い」という間接的な形でしか関係をとらえることができません.
- また, LDA は文書に含まれる各単語ごとにトピックを推定するため, 精密なモデル化ができる一方で, 大規模なコーパスでは計算量が大きくなってしまうという問題があります.

ここで, LDA で文書の $\boldsymbol{\theta}$ がわかっている場合, その文書での単語 w の確率 $p(w|\boldsymbol{\theta})$ は, トピック z を考えて周辺化することで, 式(5.81)でもみたように

∗51　[218]では $\boldsymbol{\theta}$ から直接 y への回帰モデルを求めていますが, 1.4 節で説明したようにこれはあまり適切ではなく, $-\log \boldsymbol{\theta}$ で情報量に変換してから回帰モデルを計算した方がよいでしょう.

∗52　`http://chasen.org/~daiti-m/lectures/H24-TopicModel/`

$$p(w|\boldsymbol{\theta}) = \sum_z p(w,z|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(w|z=k)p(z=k|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(w|k)\theta_k \tag{5.86}$$

と書けることに注意しましょう. 以下では数学的な便宜のため, 文書–単語行列の縦横を入れ替えて縦軸を単語, 横軸を文書として説明します [*53]. 上の確率を V 個の単語について縦に並べれば,

$$\underbrace{\begin{pmatrix} p(w_1|\boldsymbol{\theta}) \\ p(w_2|\boldsymbol{\theta}) \\ p(w_3|\boldsymbol{\theta}) \\ \vdots \\ p(w_V|\boldsymbol{\theta}) \end{pmatrix}}_{\mathbf{p}} = \underbrace{\begin{pmatrix} p(w_1|1) & p(w_1|2) & \cdots & p(w_1|K) \\ p(w_2|1) & p(w_2|2) & \cdots & p(w_2|K) \\ p(w_3|1) & p(w_3|2) & \cdots & p(w_3|K) \\ \vdots & & & \vdots \\ p(w_V|1) & p(w_V|2) & \cdots & p(w_V|K) \end{pmatrix}}_{\mathbf{B}} \underbrace{\begin{pmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_K \end{pmatrix}}_{\boldsymbol{\theta}} \tag{5.87}$$

と書くことができ, $\boldsymbol{\beta}_k$ を並べた行列 $\mathbf{B}=(\boldsymbol{\beta}_1, \boldsymbol{\beta}_2, \cdots, \boldsymbol{\beta}_K)$ を使って $\mathbf{p}=\mathbf{B}\boldsymbol{\theta}$ と表すことができます. 式(5.87)をさらに N 個の文書について横に並べると, n 番目の文書の $\boldsymbol{\theta}$ を $\boldsymbol{\theta}_n$ として

(5.88)

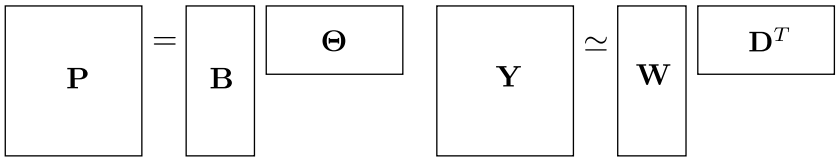
$$\underbrace{\begin{pmatrix} p(w_1|\boldsymbol{\theta}_1) & \cdots & p(w_1|\boldsymbol{\theta}_N) \\ p(w_2|\boldsymbol{\theta}_1) & \cdots & p(w_2|\boldsymbol{\theta}_N) \\ p(w_3|\boldsymbol{\theta}_1) & \cdots & p(w_3|\boldsymbol{\theta}_N) \\ \vdots & & \vdots \\ p(w_V|\boldsymbol{\theta}_1) & \cdots & p(w_V|\boldsymbol{\theta}_N) \end{pmatrix}}_{\mathbf{P}} = \underbrace{\begin{pmatrix} p(w_1|1) & \cdots & p(w_1|K) \\ p(w_2|1) & \cdots & p(w_2|K) \\ p(w_3|1) & \cdots & p(w_3|K) \\ \vdots & & \vdots \\ p(w_V|1) & \cdots & p(w_V|K) \end{pmatrix}}_{\mathbf{B}} \underbrace{\begin{pmatrix} \theta_{11} & \theta_{21} & \cdots & \theta_{N1} \\ \theta_{12} & \theta_{22} & \cdots & \theta_{N2} \\ \vdots & & & \vdots \\ \theta_{1K} & \theta_{2K} & \cdots & \theta_{NK} \end{pmatrix}}_{\boldsymbol{\Theta}}$$

となり, 図 5.31(a) に示したように行列形式で

$$\mathbf{Y} \sim \mathbf{P}, \quad \mathbf{P}=\mathbf{B}\boldsymbol{\Theta} \tag{5.89}$$

と表すことができます. つまり LDA は, $\mathbf{P}$ と同じ大きさの単語–文書行列 $\mathbf{Y}$ が

*53 Python の Numpy では, 行列の要素は標準では内部的に行方向が先に格納されるため (row-major といいます), これまで説明した文書–単語行列の形が実装上も有利です.



(a) LDA および PLSI　　(b) LSI

図 5.31: 行列分解による文書モデル. 左辺の行列を, 右辺の二つの行列の積で近似しています. このとき, 右辺の行列の各行と各列が「単語ベクトル」および「文書ベクトル」になります.

多項分布に従って $\mathbf{Y} \sim \mathbf{P}$ と生成されたと考え, この $\mathbf{P}$ を式(5.89) のように分解する**行列分解**を教師なし学習している, と考えることができるわけです. ここで $\sim$ とは, 観測値 $\mathbf{Y}$ の各列が $\mathbf{P}$ の同じ列をパラメータとして多項分布で生成されたことを表します. *54

式(5.88)で, 確率分布の形で各文書の $\boldsymbol{\theta}_n$ は文書 n の「埋め込み」, $\mathbf{B}$ の各行 $\phi(w) = (p(w|1), p(w|2), \cdots, p(w|K))$ も単語 w の一種の「埋め込み」と考えることができます. しかし, $\boldsymbol{\theta}$ も $\phi(w)$ も負になることができず, 縦横に和が 1 になる必要もあるため, この行列分解には大きな制約が伴います.

それならば, いっそのこと $\mathbf{Y}$ を図 5.31(b) のように直接,

$$\mathbf{Y} \simeq \mathbf{W}\mathbf{D}^T \tag{5.90}$$

と, 一般の実行列 $\mathbf{W}, \mathbf{D}$ に行列分解 (156 ページ) すれば, $\mathbf{W}$ と $\mathbf{D}$ の各行は制約のない単語ベクトル, 文書ベクトルになるのではないでしょうか. この考えに基づくのが, 情報検索の分野で 1990 年に提案された **LSI** (Latent Semantic Indexing) [222]でした. LSI では, 頻度 $\mathbf{Y}$ に含まれる頻度 $Y(w,d)$ をそのまま使うとほとんどを機能語が占めてしまうため, $\widetilde{Y}(w,d) = \mathrm{tfidf}\,(Y(w,d))$ のように, この後で説明する **tf.idf** のような重みづけを行ってから, 式(5.90)のように行列分解を計算して文書ベクトル・単語ベクトルを求めます *55.

---

*54　$\boldsymbol{\Theta}$ と $\mathbf{B}$ はともに非負ですから, これは非負値行列因子分解 (NMF) の一種とみなすことができます. Lee らが 2000 年に提案した NMF [219]は, 統計的には頻度がポアソン分布に従うことに対応しています. ポアソン分布はスケールが文書の長さに依存してしまい, 正規化することで多項分布となるため, 式(5.89)はその一般化といえます. NMF とテキストの Gamma–Poisson モデル (GaP) [220]については, 筆者のメモ[221]も参照してください.

*55　深層学習で文書ベクトルを求める方法としては, この後で説明する Doc2Vec や Sentence-

ただし, LSI では tf.idf のような単語の重みづけ法がアドホックで, 得られるベクトルの数学的な意味づけが弱いという大きな欠点がありました. そこで, LSI を多項分布による確率モデルとして式(5.89)のようにとらえ直した, **PLSI** (Probabilistic Latent Semantic Indexing) という革新的なモデルが 1999 年に提案され[223], それが 2001 年にベイズ化されて LDA になった[199]という歴史があります. 実験的にも, パープレキシティで測った性能は LDA>PLSI>LSI の順に高いことがわかっており[185], テキストをアドホックでなく, 確率的に考えることの重要性を示しています.

5.5.1 文書ベクトルと Doc2Vec

ただし, 実行列による行列分解自体に意味がないわけではありません. 皆さんは図 5.31 のような行列分解は, 本書でこれまでに覚えがあるのではないでしょうか. これは, 3 章の図 3.33 でみた, 行列分解によるニューラル単語ベクトルの学習

$$\mathbf{X} \simeq \mathbf{W}\mathbf{C}^T \tag{5.91}$$

と, ほとんど同じ形をしています. 単語ベクトルの場合, もとの共起行列 $\mathbf{X}$ の要素 $X(w,c)$ は, 単語 w とその周辺に出現した文脈語 c との非負自己相互情報量 (PPMI)

$$X(w,c) = \mathrm{PPMI}(w,c) = \max\left(\log\frac{p(w,c)}{p(w)p(c)}, 0\right)$$

でした (式(3.112)). そこで, 単語–文書行列の場合は単語 w とそれが出現した文書 d の PPMI を用いて,

$$X(w,d) = \mathrm{PPMI}\,(w,d) = \max\left(\log\frac{p(w,d)}{p(w)p(d)}, 0\right) \tag{5.92}$$

を並べた行列 $\mathbf{X}$ を作り, これを図 5.32 のように

$$\mathbf{X} \simeq \mathbf{W}\mathbf{D}^T \tag{5.93}$$

BERT といった多数の手法がありますが, これらは内部的にはいずれも, 本章と同様に単語ベクトルを利用したものです.

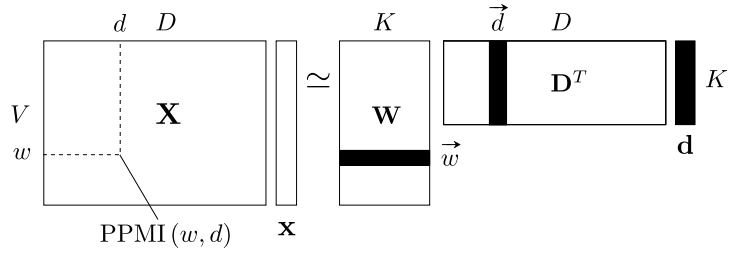


図 5.32: DocVec による文書ベクトルと単語ベクトルの計算. 右辺の $\vec{w}$ と $\vec{d}$ の内積で左辺の PPMI (w, d) を近似し, これは Doc2Vec (Word2Vec) の学習と数学的に等価です.

と行列分解すれば, 行列 $\mathbf{W}$ の各行 $\vec{w}$ は Word2Vec と数学的に等価な「ニューラル単語ベクトル」に, 行列 $\mathbf{D}$ の各行 $\vec{d}$ は「ニューラル文書ベクトル」になるのではないでしょうか [*56]. 式(5.93)の分解は, 3.5.4 節の単語ベクトルの計算での式(3.116)および式(3.117)と同様に, $\mathbf{X}$ を上位 K 個の特異値 ($=\mathbf{X}\mathbf{X}^T$ の固有値) を使って

$$\mathbf{X} \simeq \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^T = \underbrace{\left(\mathbf{U}_K \mathbf{S}_K^{1/2}\right)}_{\mathbf{W}} \underbrace{\left(\mathbf{V}_K \mathbf{S}_K^{1/2}\right)}_{\mathbf{D}}^T \tag{5.94}$$

と特異値分解し,

$$\mathbf{W} = \mathbf{U}_K \mathbf{S}_K^{1/2}, \quad \mathbf{D} = \mathbf{V}_K \mathbf{S}_K^{1/2} \tag{5.95}$$

とすれば計算することができます.

実際に, Mikolov らが Word2Vec の後に提案し, よく使われている **Doc2Vec** [224]は文書ベクトルから文書に含まれる単語を予測するモデルで, 単語ベクトルのスキップグラム (3.5.3 節) と同じ目的関数をしており, 式(5.93)と等価です. しかし, 確率的勾配法による繰り返し計算で近似を行う Doc2Vec に対して, 式(5.95)から得られる文書ベクトル (本書ではこれを **DocVec** とよぶことにします) はこの後で説明するように線形代数で高速に解くことができ, 数学的な最適解が求まるために, 性能も高いことを筆者が確かめています[225].

---

*56　すなわち, LSI に足りなかったのは適切な単語の重みづけと, それを支える背後の理論だったということです. 156 ページの脚注で説明したように, 実験的には PPMI による単語重みづけが高性能であることは, 深層学習以前に一部ではすでに知られていました.

アルゴリズム 8: 文書ベクトル (DocVec) の計算アルゴリズム.

1: 式(5.96)を転置した文書–単語行列 $\mathbf{X}$ を疎行列フォーマットで作成する.
2: $\mathbf{U}, \mathbf{S}, \mathbf{V} = \mathrm{svds}\,(\mathbf{X}, K)$ と K 次元に特異値分解する.
3: 文書ベクトル行列 $\mathbf{D} = \mathbf{U}\mathbf{S}^{1/2}$, 単語ベクトル行列 $\mathbf{W} = \mathbf{V}^T\mathbf{S}^{1/2}$ を求める.

図 5.33: 特異値分解による文書ベクトル (DocVec) の計算アルゴリズム. Python は行列を横方向に格納するため, ここでは式(5.93)の両辺を転置した $\mathbf{X}^T \simeq \mathbf{D}\mathbf{W}^T$ を計算しています.

なお, $\mathbf{X}$ の各要素である式(5.92)は, ベイズの定理から $p(w\,|\,d) = p(w,d)/p(d)$ ですから,

$$\mathrm{PPMI}\,(w,d) = \max\left(\log\frac{p(w,d)}{p(w)p(d)}, 0\right) = \max\left(\log\frac{p(w\,|\,d)}{p(w)}, 0\right) \tag{5.96}$$

で求めることができます. すなわち, この値は「文書 d の中で単語 w が通常より何倍多く出現したのか」という比の対数となっています. d の中での w (たとえば “the”) の出現確率が平均的な確率とほぼ同じならば, $p(w\,|\,d) \simeq p(w)$ ですから, 式(5.96)の比は

$$\log\frac{p(w\,|\,d)}{p(w)} \simeq \log 1 = 0$$

となることに注意してください. 機能語はどの文書でも確率がほぼ一定なので, こうすれば**「ストップワード」を準備しなくても, 機能語の PPMI はほぼ 0 になる**ことがわかります.

DocVec の学習アルゴリズムを, 図 5.33 に示しました. この計算は, サポートサイトの `docvec.py` で行うことができます [*57]. 図 5.32 からわかるように, このとき $\mathbf{D}$ の各行として文書ベクトル $\vec{d}$ が, $\mathbf{W}$ の各行として単語ベクトル $\vec{w}$ が得られることに注意してください. 図 5.34 に示したように, これらのベクトルは**同じ K 次元の潜在空間に存在しており**,

$$\mathrm{PPMI}\,(w,d) \simeq \vec{w}\cdot\vec{d} \tag{5.97}$$

*57 疎行列の特異値分解を行う SciPy の `svds()` を使う場合, 特異値が大きい順ではなく, 小さい順に返されますので注意してください. 標準では, 次元が重要な順の逆に並ぶことになります.

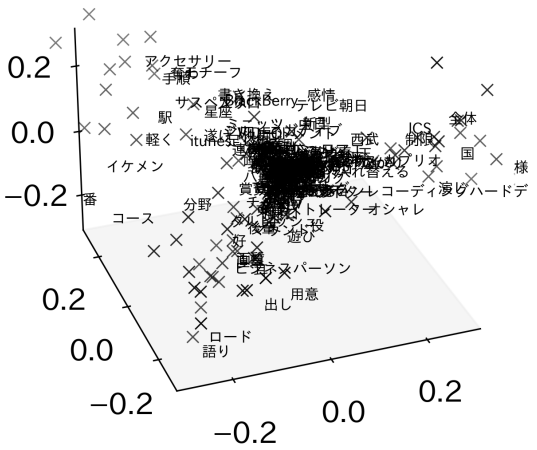


図 5.34: livedoor コーパスから DocVec で計算した文書ベクトル (×印) と単語ベクトルの一部. 3.5.4 節で議論したように, 最大の特異値に対応する次元は全体のバイアスを表しているため, ここでは特異値の大きい方から 2,3,4 次元目の値でプロットしました. これから, livedoor コーパスの文書ベクトル・単語ベクトルはこの次元の範囲では, 大きく「女性軸」(左上), 「電気製品軸」(右), 「一般軸」(左下) に沿って分布していることがわかります. ただし, 座標軸が必ずしもそれらに沿ってとられているわけではありません.

となるように文書ベクトル $\vec{d}$, 単語ベクトル $\vec{w}$ が学習されることになります.

文書ベクトルの計算　実際に実験してみましょう. サポートサイトにある `docvec.py` を使うと, livedoor コーパスのデータ `livedoor.dat` について次のように実行すれば, $K = 200$ 次元の文書ベクトルと単語ベクトルが計算できます.

```
   % docvec.py -K 200 -d livedoor.lex livedoor.dat model.docvec
⇒  using dictionary: livedoor.lex
   livedoor.dat: K = 200, output = docvec.livedoor.K200
   parsing data..
   creating sparse matrix..
   computing data vectors..
   done.
   writing model to model.docvec.. done.
```

この計算は, 執筆時の環境では 14.6 秒で終了しました. これから, たとえば 200 番目の文書に似た文書ベクトルを検索したい場合は, コーパスの実際の中身が書かれているテキスト livedoor.txt (240 ページ) を同時に与えて, 次のように実行します.

```
   % docvec-similar.py docvec.model livedoor.txt 200
⇒ loading model from docvec.model.. done.
    1.0000 dokujo-tsushin 30 歳を過ぎた大人の肌へ、世界が認めた美容液「」
    0.6536 dokujo-tsushin 女性も驚愕!?スキンケアしていない人は 70%と男
    0.6530 peachy         あなたの見た目年齢を上げているのは“シミ”だった
    0.6492 peachy         自宅でできるでふっくらお肌を目指そう!毎朝のメイ
    0.6315 peachy         日本初!DHC、“10 倍濃度”の Q10 シリーズを
    0.6148 kaden-channel  ライオン、毛と音波振動でくすみを落とす「プラチア
    0.6104 peachy         男性の 4 割が“すっぴん”にがっかり!最強のすっぴ
    0.5819 peachy         “究極のクリーム”で 10 年後も 20 年後もずっと綺
```

一番上の文書は自分自身 (類似度は $\cos 0 = 1$) ですが, ジャンルを超えて内容的に近い文書が検索できていることがわかります. 実装については, スクリプトの中身を読んでみてください.

キーワードによる検索 それでは, あるキーワード集合に近い文書を探したい場合はどうすればいいでしょうか. 実は, これは自明ではありません.[*58] というのは, いま探したい “映画 東京” のような検索語だけを含む文書は学習データには存在しないからです.

しかし, キーワード集合に対応する仮想的な「文書ベクトル」$\mathbf{d}$ が計算できれば, 学習した文書ベクトルと $\mathbf{d}$ を上記のように比べればよいでしょう. ここで, 図 5.32 では, 左辺の観測データ $\mathbf{X}$ は常に式(5.92)の PPMI の形で与えられることに注意してください. この値が 0 のとき, 文書内での単語の確率は標準的な確率と等しいか小さいことを意味します. よって, 検索したい語の PPMI をたとえば 1 にすれば, これは仮想的な「文書」の中で確率が $e^1 \simeq 2.72$ 倍高いことを意味します. そこで, キーワード集合に含まれる語について 1, 他は 0 になる V 次元の縦ベクトル $\mathbf{x}$ を用意し, 図 5.32 に表したように

---

*58 簡易的には, キーワードに対応する単語ベクトルの平均を計算して文書ベクトルと比較するという方法が考えられますが, モデル上これが最適である保証はありません. 実際, この後で計算して対応する列の和をとる行列 $(\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T$ は, $\mathbf{W}$ 自体とは異なったものです.

$$\mathbf{x} \simeq \mathbf{W}\mathbf{d} \tag{5.98}$$

が成り立つ文書ベクトル $\mathbf{d}$ を求めればよい, ということになります.

特異値分解は両辺の二乗誤差を最小化する方法ですので, 最小二乗の意味で式(5.98)が成り立つ $\mathbf{d}$ を求めるには, 単純に最適化を行うこともできますが, 式(5.98)はよく知られた線形回帰モデル (OLS) ですから, $\mathbf{d}$ の最適解は

$$\mathbf{d}^* = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{x} \tag{5.99}$$

で与えられます[12, §1.2]. 式(5.95)より $\mathbf{W}=\mathbf{U}_K\mathbf{S}_K^{1/2}$ でしたから, 簡単のためにこれを $\mathbf{U}\mathbf{S}^{1/2}$ と書くと, 上の式は

$$\begin{aligned}\mathbf{d}^* &= ((\mathbf{U}\mathbf{S}^{1/2})^T\mathbf{U}\mathbf{S}^{1/2})^{-1}(\mathbf{U}\mathbf{S}^{1/2})^T\mathbf{x} \\ &= (\mathbf{S}^{1/2}\mathbf{U}^T\mathbf{U}\mathbf{S}^{1/2})^{-1}\mathbf{S}^{1/2}\mathbf{U}^T\mathbf{x} \\ &= \mathbf{S}^{-1}\mathbf{S}^{1/2}\mathbf{U}^T\mathbf{x} = \mathbf{S}^{-1/2}\mathbf{U}^T\mathbf{x}\end{aligned}$$

となります[*59]. 2 行目から 3 行目では, $\mathbf{S}$ が対角行列であること, および特異値分解で得られる $\mathbf{U}$ は直交行列なので $\mathbf{U}^T\mathbf{U}=\mathbf{I}$ であることを用いました. よって, あらかじめ回帰行列 $\mathbf{R}=\mathbf{S}_K^{-1/2}\mathbf{U}_K^T$ を計算しておけば, 式(5.99)は

$$\mathbf{d}^* = \mathbf{R}\mathbf{x} \tag{5.100}$$

と, 一瞬で最適解が求められます[*60]. なお, キーワードではなく新しい文書自体が与えられている場合は, 式(5.96)から PPMI を計算して $\mathbf{x}$ を作れば, 同様にして最適な文書ベクトル $\mathbf{d}^*$ を求めることができます.

この方法で livedoor コーパスの文書について, 意味を考慮したキーワード検索を行った例は次のようになります. ニューラル単語ベクトルを介しているため, 検索語自体が出現していなくても, 意味が似ていれば (=単語ベクトルが近ければ) 文書が検索できることに注意してください.

```
% docvec.py -K 200 -R -d livedoor.lex livedoor.dat model.docvec-R
```

---

*59　$\mathbf{W}=\mathbf{U}_K\mathbf{S}_K^{1/2}$ を式(5.98)に代入すると $\mathbf{x}\simeq\mathbf{U}_K\mathbf{S}_K^{1/2}\mathbf{d}$ となり, フルランクの SVD に戻って考えれば, これから $\mathbf{d}^*=\mathbf{S}_K^{-1/2}\mathbf{U}_K^T\mathbf{x}$ を直接導くこともできます.

*60　$\mathbf{y}$ の要素はほとんどが 0 ですから, 式(5.100)の計算は実際には, $\mathbf{R}$ のうち $\mathbf{x}$ の要素が 1 の列を足すだけの操作になり, `mmap()` などで $\mathbf{R}$ をディスクに保存すれば, 空間計算量も最小限に抑えることができます.

```
# -R をつけて実行し，回帰行列を事前に計算しておく
% docvec-search.py model.docvec-R livedoor.txt 映画 東京
⇒ loading model from model.docvec-R.. done.
keyword: 映画 東京
 0.0158 movie-enter    この夏、東京スカイツリーが映画に 2008 年 7 月の
 0.0152 movie-enter    小栗旬は“使えない若者”、映画『キツツキと雨』の
 0.0152 movie-enter    スパイダーマンが地上 75 メートルの通天閣を『アメ
 0.0149 peachy         【スナップレポート】東京ガールズコレクション 2011
 0.0149 movie-enter    食べて、で、映画を観れる『東京ごはん映画祭』が今
 0.0149 peachy         【スナップレポート】東京ガールズコレクション 2011
 0.0148 movie-enter    基礎から勉強しよう!初心者でもわかる「東京国際映
 0.0144 movie-enter    三池監督が映画『一命』について「満島ひかりがいろ
 0.0141 movie-enter    東京国際映画祭、『最強のふたり』がグランプリを受
```

メモ：tf.idf と単語の重みづけ

自然言語処理の多くの場面で, “日” “方法” のように意味の弱い語には低い重みを, “ペプチド” “紺碧” のような意味の強い語には高い重みを与えたい場合があります. このための古典的な単語重みとして, **tf.idf** [226]というヒューリスティックが知られています.

意味の弱い言葉の特徴は,「どんな文書にもまんべんなく現れる」ということでしょう. これに対して, 意味の強い言葉は特定の話題と結びついており, 全体のごく一部の文書にしか現れないのが特徴です. よって, N 個の文書の中で, ある単語 w が一度以上出現した文書の数を $\mathrm{df}(w)$ とおくと (これを**文書頻度** (document frequency) といいます)[*61], 全体に対する割合 (文書確率)

$$p = \frac{\mathrm{df}(w)}{N}$$

が 1 に近いほど w の意味は弱く, 0 に近いほど意味が強いと考えられます. $\log 1 = 0$ ですから, この情報量 (式(2.64)) をとった

単語 w	$\mathrm{df}(w)/N$	$\mathrm{idf}(w)$			
日	0.7003	0.3563	ソニー	0.0300	3.5075
人	0.5640	0.5726	知識	0.0272	3.6028
年	0.5014	0.6904	芸能	0.0245	3.7081
的	0.4632	0.7696	夏休み	0.0191	3.9595
日本	0.3651	1.0075	手作り	0.0163	4.1136
時間	0.2371	1.4395	画質	0.0136	4.2959
使用	0.1253	2.0767	新曲	0.0109	4.5191
姿	0.0899	2.4089	真っ白	0.0082	4.8067
NTT	0.0790	2.5381	危機	0.0082	4.8067
説明	0.0654	2.7273	豪雨	0.0054	5.2122
Web	0.0327	3.4205	愛着	0.0054	5.2122

表 5.11: livedoor コーパスから計算した, 単語 w の文書確率 $\mathrm{df}(w)/N$ と $\mathrm{idf}(w) = \log N/\mathrm{df}(w)$ (抜粋). “日” のような語は全体の 7 割の文書に出現しているため idf は小さく, 一方で “豪雨” のような語は全体の 1%未満の文書にしか出現していないため, idf が大きいことがわかります.

*61　$n(i,w)$ を使うと, 数学的には $\mathrm{df}(w) = \sum_{i=1}^{N} \mathbb{I}(n(i,w) > 0)$ と表すことができます.

$$\text{idf}(w) = -\log p = \log \frac{N}{\text{df}(w)} \tag{5.101}$$

を, 単語 w の意味的な重みと考えることができます. df が分母にあることから, これを**逆文書頻度** (inverse document frequency, idf) [227] といいます. 表 5.11 に, livedoor コーパスで計算した単語の p と idf の例を示しました.

いっぽう, 特定の文書 d_i の中では, 単語 w の出現回数 $n(i,w)$ が大きいほど意味は強いでしょう. ただしその影響は線形ではなく, 実質的な強さはその対数くらいだと考えられます (ウェーバー=フェヒナーの法則). つまり, 「ソニー」が 50 回出現したからといってその情報量が 50 倍あるわけではなく, 効果は $\log 50 \simeq 3.91$ 倍に比例する程度だということです. よって, $\log 1 = 0$ となることを避けて, しばしば

$$\text{tf}(n) = 1 + \log n \tag{5.102}$$

のように定義されます. これを**用語頻度** (term frequency) とよびます. tf と idf を組み合わせると, 文書 d_i での単語 w の頻度 $n(i,w)$ を

$$\text{tf}(n(i,w)) \cdot \text{idf}(w) = (1 + \log n(i,w)) \cdot \log \frac{N}{\text{df}(w)} \tag{5.103}$$

と変換すれば, 適切な単語重みになると考えられます. この重みづけを **tf.idf** といいます.

実際には対数の底に任意性があるなど, いくつかのバリエーションがあります[15, §15.2.2]. tf.idf は有効に働くヒューリスティックで, idf には最大エントロピー法による理論的な説明もありますが[228], 自然言語処理のタスクのために tf.idf が最適であるという保証はなく, 現代的には 5.2.1 節の PMI や, 4.2.2 節の SIF による重みづけの方がそれぞれの場合で数学的な最適性があり, 効果的です.

5.5.2　単語ベクトル/文書ベクトルの解釈

こうして得られた文書ベクトルは高速に計算でき, 数学的に Word2Vec(Doc2Vec) と同じニューラル文書ベクトルとなっているため, 高い性能を持っています. 唯一の欠点は, K 個の次元が LDA のようにトピックとして解釈ができないということでしょう. たとえば, 上の実験で得られた単語ベクトルを並べた行列 $\mathbf{W}$ について, その 1 次元目, 2 次元目, …の値が大きい単語を求めると表 5.12 のようになり, ここには強い規則性は見出せそうにありません.

考えてみるとこれは当然で, 式(5.93)による行列分解は式(5.97)のように内積だけを問題にしているため, 空間全体を任意に回転しても, 図 5.35 のように 2 つのベクトルの間の内積は同じになるからです. 数学的には, ある直交行列 $\mathbf{R}$ をとって $\widetilde{\mathbf{D}} = \mathbf{DR}$, $\widetilde{\mathbf{W}} = \mathbf{WR}$ とベクトル全体を回転したとき, 式(5.93)の右辺は

$$\widetilde{\mathbf{D}}\widetilde{\mathbf{W}}^T = \mathbf{DR}(\mathbf{WR})^T = \mathbf{D}\underbrace{\mathbf{RR}^T}_{=\mathbf{I}}\mathbf{W}^T = \mathbf{DW}^T \tag{5.104}$$

となり, もとと等しくなります.

すなわち, 解釈のためには図 5.35 に示したように, 単語ベクトルや文書ベクトルがちょうど軸の近くに配置されるような回転 $\mathbf{R}$ を見つける必要がありま

次元 1		次元 2		次元 3		次元 4	
級	0.6090	自身	0.6138	Watch	0.7628	再生	0.7540
節電	0.5832	イベント	0.5650	Sports	0.6780	ホラー	0.5644
全	0.5279	クリスマス	0.5647	婚	0.5346	恐怖	0.5515
電力	0.5185	http	0.5261	活	0.5325	音楽	0.5425
シリーズ	0.4905	作っ	0.5027	答え	0.5069	デビュー	0.4848
AKB	0.4835	城	0.4955	音楽	0.5028	PC	0.4742
母親	0.4638	シーズン	0.4949	佐	0.4505	現象	0.4396
通話	0.4564	テレビ	0.4719	曲	0.4487	娘	0.4229
スマ	0.4509	婦	0.4476	学校	0.4450	ロンドン	0.4222
出展	0.4390	超	0.4430	枝	0.4409	YouTube	0.4201
家族	0.4374	家政	0.4391	調査	0.4366	必ず	0.4186
額	0.4268	www	0.4380	選手	0.4173	韓国	0.4184

表 5.12: livedoor コーパスから DocVec で計算した単語ベクトルの各次元の値が大きい単語 (一部). もとのままでは, 各次元に明確な意味は見出せそうにありません.

す. こうした方法として, 心理学ではバリマックス回転のような方法が知られています[229]. しかし, これは言語のように数百次元以上にもなる高次元の場合にはあまりうまくいかず[230], 最近, 京都大学の下平らにより, **独立成分分析** (Independent Component Analysis, **ICA**) を適用することで解釈が容易な軸が, しかも言語横断的に見つかることが示されました[231]. ICA はデータを線形変換して, 各次元が可能な限り統計的に独立となるような座標軸を発見する方法です [*62]. たとえば図 5.36 では, PCA(主成分分析) ではデータの分散を最大にするように真ん中のような座標軸がとられてしまいますが, ICA ではデータの分布を独立な軸の積で説明する右のような座標軸を計算することができます.

「統計的に独立」とは, 2 章の式(2.21) でみたように, 確率変数 x と y の同時確率が $p(x,y)=p(x)p(y)$ のように周辺確率の積に分解できることでした. われわれの場合, たとえば式(5.93)で得られた, 単語ベクトルを並べた行列 $\mathbf{W}$ を行列 $\mathbf{A}$ を使って $\mathbf{S}=\mathbf{WA}$ と線形変換したとき, $\mathbf{S}$ の各行ベクトル $\mathbf{s}=(s_1,s_2,\cdots,s_K)$ について, 分解

$$p(s_1,s_2,\cdots,s_K)=p(s_1)p(s_2)\cdots p(s_K) \tag{5.105}$$

が可能な限り成り立つような行列 $\mathbf{A}$ を求めることになります [*63].

こうした $\mathbf{A}$ は, 機械学習の分野で ICA を定式化したフィンランドの Hyvärinen

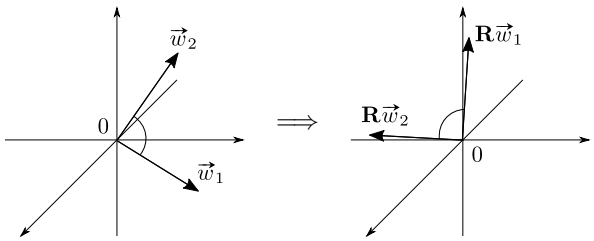


図 5.35: 単語ベクトルの回転. 単語ベクトルを直交行列 $\mathbf{R}$ で回転しても, 二つの単語ベクトルのなす角度は不変です. 適切な回転 $\mathbf{R}$ を求めることで, 単語ベクトルを軸に沿った形で, よりわかりやすく解釈することができます.

---

*62 169 ページで説明した白色化はデータを無相関にする方法ですが, 独立とは無相関を含んでおり, それよりも強い条件になります.

*63 ICA は実際には, データの白色化を行ってから $\mathbf{R}$ による回転を行うことと等価で, 白色化行列を $\mathbf{B}$ とおくと $\mathbf{A}=\mathbf{BR}$ になります.

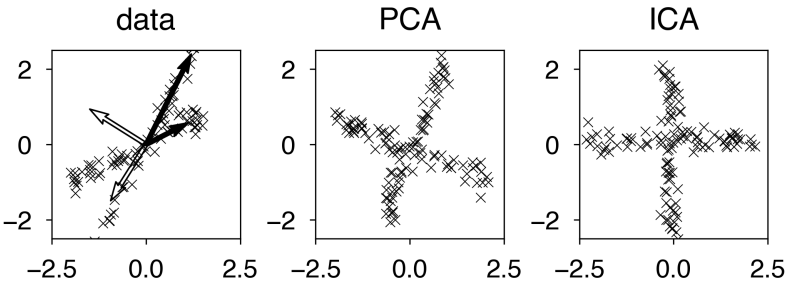


図 5.36: ICA の概要図. PCA(主成分分析) では白矢印のように, データ (×印) の分散を全体的に説明する軸がとられてしまいますが, ICA(独立成分分析) では黒矢印のように, データの分布を独立な軸の積として説明する軸が求められているのがわかります.

によるパッケージ, `FastICA`[*64] で計算することができます. Python では,

```
from sklearn.decomposition import FastICA # ICA の計算
from scipy.stats import skew              # 歪度の計算
import numpy as np
def ica (X):
    X = X - np.mean (X, axis=0)
    analyzer = FastICA (whiten="arbitrary-variance")
    S = analyzer.fit_transform (X)
    A = analyzer.components_
    # sort by skewness
    N,D = S.shape
    skews = np.abs (skew (S, axis=0)) # 0=Gaussian
    index = map (lambda x: x[1],
                 sorted (zip(skews, np.arange(D)),
                         key=lambda x: x[0], reverse=True))
    return S[:,list(index)], A
```

で $\mathbf{S}$ および $\mathbf{A}$ を求めることができます. 多くの信号が混ざると, 中心極限定理によって分布はガウス分布に近づくことから, 逆に ICA で分解した軸では, 各次元の周辺分布は非ガウス的になります. その度合いは, 期待値 μ, 標準偏差 σ をもつ確率変数 X の**歪度**(わいど)

$$\delta = \mathbb{E}[(X-\mu)^3]/\sigma^3 \tag{5.106}$$

*64 `http://research.ics.aalto.fi/ica/fastica/`

次元 3		次元 5		次元 6		次元 10	
iPhone	0.0224	apps	0.0853	ビジネス	0.0180	等	0.0272
クリック	0.0185	google	0.0851	経営	0.0174	由里子	0.0269
ブック	0.0163	要件	0.0848	成功	0.0172	吉	0.0268
電子	0.0160	store	0.0847	キャリア	0.0164	愛	0.0265
術	0.0160	play	0.0842	オフィスエム	0.0155	サイト	0.0260
アップ	0.0152	ANDROID	0.0821	笑	0.0147	幸せ	0.0246
サイト	0.0150	details	0.0820	話し	0.0144	果たし	0.0243
携帯	0.0142	Play	0.0779	管理	0.0139	公式	0.0242
既報	0.0135	Google	0.0600	スキル	0.0132	務め	0.0234
背面	0.0134	Hisumi	0.0565	戦略	0.0129	篇	0.0233
ライフ	0.0123	以上	0.0421	力	0.0129	リアル	0.0224
iPad	0.0123	Android	0.0420	重要	0.0124	女優	0.0223

次元 15		次元 19		次元 28		次元 39	
高画質	0.0122	サッカー	0.2559	ロードショー	0.0428	スタイル	0.0218
デジタル	0.0105	代表	0.2401	全国	0.0425	オシャレ	0.0197
実現	0.0095	戦	0.2193	女優	0.0237	着	0.0171
操作	0.0093	試合	0.2175	決意	0.0206	ファッション	0.0170
ソニー	0.0084	杯	0.1897	土	0.0200	楽しむ	0.0166
ズーム	0.0082	W	0.1591	実力	0.0199	シンプル	0.0164
進化	0.0081	チーム	0.1532	最強	0.0198	ダイエット	0.0153
保存	0.0080	ゴール	0.1410	黄金	0.0194	味わい	0.0153
シーン	0.0080	日本	0.1280	絆	0.0193	体重	0.0151
新	0.0079	予選	0.1277	祭	0.0187	食事	0.0151
軽量	0.0079	選手	0.1261	ベルセルク	0.0186	誰	0.0147
レス	0.0079	リーグ	0.1239	TOHO	0.0184	運動	0.0145

表 5.13: ICA で変換した単語ベクトルの各次元の値が大きい単語 (抜粋). 次元は歪度の絶対値の大きい順に並んでいます. 表 5.12 と比べて, ほとんどトピックモデルのような, 意味的な軸が学習されていることがわかります. 次元 10 の意味については, 表 5.14 を参照してください.

で表すことができます[232]. ガウス分布では δ は 0 で, $\pm$ になるほど左右の裾が広くなることが知られています. ICA は PCA と異なり, 独立性の高い軸から求まるとは限りませんので, 上のコードでは δ の絶対値を用いて, 非ガウス性の高い順に次元を並び換えています. livedoor コーパスから計算した単語ベクトルについて, ICA で変換した各次元の値の大きい単語を表 5.13 に示しました. ほとんどトピックモデルのような, 解釈性の高い意味的な軸が求められていることがわかります.

ただしトピックモデルと異なり, これはベクトル空間の「軸」ですので, 負の

次元 3		次元 6		次元 10		次元 19	
apps	−0.2364	お答え	−0.2046	妖	−0.2595	フィギュア	−0.0405
store	−0.2363	辛口	−0.2041	ヶ	−0.2562	スケート	−0.0403
details	−0.2362	悩め	−0.2026	巻	−0.2316	選手権	−0.0364
play	−0.2300	説教	−0.2018	劇場	−0.1994	演技	−0.0312
google	−0.2299	尽き	−0.1938	勅使河原	−0.1992	克也	−0.0268
要件	−0.2225	早	−0.1856	妖怪	−0.1967	浅田	−0.0265
ANDROID	−0.2187	面白く	−0.1820	栄華	−0.1961	真央	−0.0262
id	−0.2119	姉妹	−0.1689	仙人	−0.1900	メダリスト	−0.0262
Play	−0.2006	悩み	−0.1680	お嬢様	−0.1695	野村	−0.0255
com	−0.1913	type	−0.1663	ネコ	−0.1589	ノム	−0.0246
Store	−0.1798	若手	−0.1650	話	−0.1497	バンクーバー	−0.0231
カテゴリ	−0.1682	瞬時	−0.1596	次	−0.1430	野球	−0.0225

表 5.14: ICA で変換した単語ベクトルの各次元の値が小さい単語 (抜粋). 表 5.13 と見比べると, 次元 3 では iPhone の「反対の概念」として Android が, 次元 6 ではビジネスについてお悩み相談が, 次元 19 ではサッカーについてフィギュアスケートが得られていることがわかります. 次元 10 は元データの livedoor ブログでの “いちおう妖ヶ劇場” という連載記事 [*65] に関連している軸で, 負の方にだけ意味を持っています.

方向も存在します [*66]. 表 5.14 に, 表 5.13 のいくつかの軸について値が負の単語を示しました. 単なる文書–単語の共起行列から得られたにもかかわらず, iPhone ⟷ Android, ビジネス ⟷ 悩み相談のような興味深い対立軸が教師なしで得られていることがわかります.

5.6* 確率的潜在意味スケーリング (PLSS)

前節の ICA の結果から, 文書ベクトルおよび単語ベクトルの存在する埋め込み空間には, トピックモデルのトピックに対応するようなさまざまな「軸」が存在することがわかりました. 実際に Word2Vec (これは前節で主成分分析で求めたものと, 3.5.4 節で説明したように数学的には同じものでした) の空間には, ICA で見つかるものに限らず多くの意味的な軸が存在することがわかっています[233].

特に重要なのは, 表 5.14 でみたように多くの場合, この軸は正の方向と負の

*66　空間全体をある軸に関して折り返してもベクトル間の関係は変わりませんので, 符号が正負どちらになるかに大きな意味はありません.

*65　https://news.livedoor.com/article/detail/6029862/

方向をもつ対立軸となっているということです [*66]. すると, 表 5.13 と表 5.14 で軸と向きが近い, または逆の単語を計算したように, 図 5.37 のようにこの軸上に文書ベクトルを射影し, テキストの性質をある軸に沿って 1 次元の ± の実数値として表すことができるでしょう. これを心理学の用語では, テキストの**尺度化** (scaling) といいます.

テキストを分類するのではなく, 連続値で測る尺度化は実際のさまざまな場面において有効です. たとえば, 法案を保守 (右翼) か革新 (左翼) のどちらかに分類するのは簡単でも, 実際の法案は日本ではほとんど保守 (自民党) の側から提出されており, 問題はむしろ, その法案がどれくらい保守的なのか, ということでしょう. また, 住民へのアンケートやホテルの評価に書かれたテキストを肯定・否定に分類するのは簡単ですが, 重要なのはどの意見が強い賛成・反対であり, どの意見が軽い文句なのかを見極めることでしょう. 小説家のテキストからその分裂気質を測りたいといった場合でも, 分裂症かどうかという二値分類にはあまり意味がなく (小説家の多くは分裂気質のため), 分裂気質の強さやその変化が主な興味の対象になると考えられます. [*67]

こうした尺度化を行うもっとも簡単な方法は, 単語ベクトルや文書ベクトルの与えられた軸への近さを, 前節のように cos 類似度を計算して求めることです. しかし, 3 章の図 3.39 で示したように, 埋め込みベクトルの間の cos 類似度は 0 を中心には分布せず, 値も $[-1, 1]$ の中の一部しかとりません. 別の言い方をすれば, cos 類似度はあくまで後づけの値であり, ある軸に関係しないベクトルの値が 0 となるような**尺度**としては設計されていない, ということです.

Wordfish

特に, 社会科学においてテキストの尺度化は重要な問題であるため, 政治学方法論 (Political methodology) の分野では, Laver らが 2003 年に Wordscores [235] を, Slapin らが 2008 年に政党などの連続した極性の時間変化を求める Wordfish

---

*66 ただし, Transformer で得られる埋め込み空間は注意機構のために複雑になり, こうした線形性があまり成り立たなくなっています.

*67 図 5.5 でもみたように, 分類モデルは尤度を上げるために極性を ±1 のどちらかに寄せる傾向があり, SVM のようなベクトル空間の分類器でも, 分離平面からの距離がクラスに対応する尺度とは限りません [234].

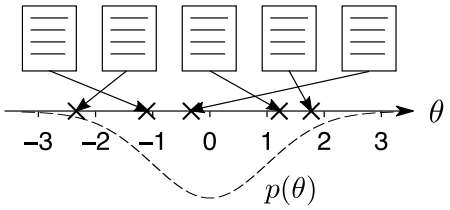


図 5.37: テキストの尺度化と潜在スケール (尺度) θ. 各文書に対して, 測りたい軸に沿った実数値 $\theta \in \mathbb{R}$ を推定します. 項目反応理論に従い, θ は標準正規分布 $\theta \sim \mathcal{N}(0,1)$ に従う潜在変数だと考えます.

[236] [*68] という方法を提案しました.

y_{itv} を政党 i が時刻 t のテキスト (たとえば選挙時の公約) で単語 v を使った回数とすると, Wordfish はデータ $Y=\{y_{itv}\}$ $(i=1,\cdots,I, t=1,\cdots,T, v=1,\cdots,V)$ の確率を, 次のようにポアソン分布 $\mathrm{Po}(y|\lambda)$ (式(4.18)) でモデル化します.

$$\begin{cases} p(Y) = \prod_{i=1}^{I}\prod_{t=1}^{T}\prod_{v=1}^{V} \mathrm{Po}(y_{itv}|\lambda_{itv}) \\ \lambda_{itv} \;\; = \exp\left(\alpha_{it} + \beta_v + \phi_v \cdot \theta_{it}\right) \end{cases} \tag{5.107}$$

ここで α_{it} はテキスト it での固定効果 (ベースライン), β_v は単語 v の固定効果で, 興味があるのは単語 v の極性軸上での位置 $\phi_v \in \mathbb{R}$ と, 政党 i の時刻 t での潜在位置 [*69] $\theta_{it} \in \mathbb{R}$ です. Wordscores は ϕ_v にあたる単語の極性「スコア」を, 極性の値を既知とした文書から計算する簡単なアルゴリズムですが, Wordfish はそれを教師なし学習として統計モデル化したものといえます.

式(5.107)は, 頻度 y_{itv} はポアソン分布 $\mathrm{Po}(\lambda)$ に従い, その期待値はテキスト it と単語 v で決まるベースライン $\alpha_{it}+\beta_v$ を, 政党 i の時刻 t での極性 θ_{it} と単語の持つ極性 ϕ_v で上下して決まる, ということを意味しています. $\theta > 0$ が右翼, $\theta < 0$ が左翼を表すとしたとき, 政党の位置と単語の符号が一致する, すなわち $\theta_{it}>0$ かつ $\phi_v>0$ (たとえば $v=$“軍備”), または $\theta_{it}<0$ かつ $\phi_v<0$ (たとえば $v=$“社会保障”) のとき y_{itv} の期待値 λ_{itv} は大きくなり, 符号が逆ならば小さくなります. $\{\alpha,\beta,\theta,\phi\}$ はすべて未知のため, 推定には 5.2.2 節で学習し

*68　`https://tutorials.quanteda.io/machine-learning/wordfish/`
*69　これを政治学の分野では, 理想点 (ideal point) といいます.

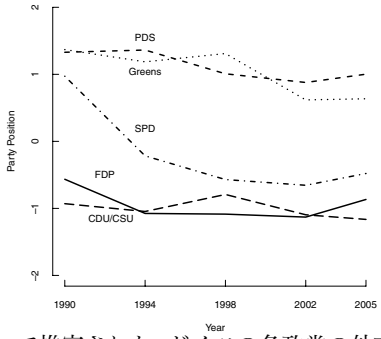


図 5.38: Wordfish によって推定された, ドイツの各政党の外交方針 θ_{it} の時間変化の例 (1990–2005) ([236]より引用). 縦軸の θ は平均が 0 の潜在変数になっています.

た EM アルゴリズムを用い, 適切な初期値から始めて, (α, θ) の推定と (β, ϕ) の推定を反復します. こうして求めたドイツの各政党の θ_{it} の時間変化を, 図 5.38 に示しました.

5.6.1 項目反応理論によるテキストの尺度化

Wordfish は与えられたテキストに対する系統的な統計モデルですが, 完全な教師なし学習のため, これによって得られる θ の極性が分析の目的と一致している, という保証はありません. 適切な結果を得るためには, 入力のテキストを注意深く選ぶ必要がありますが, そのための客観的な方法は示されておらず, また学習データの単語頻度に対するポアソン分布はテキストの長さに暗黙に依存するため, 新しいテキストでの θ を計算できないという問題もあります.

そこで, 筆者は心理統計学における**項目反応理論** (Item Response Theory, IRT) を参考に, 潜在的な極性 $\theta \in \mathbb{R}$ をもつテキストで単語 $v \in \{1, \cdots, V\}$ が出現する確率を, 次のように多項分布でモデル化しました.

$$
\begin{aligned}
p(v \mid \theta, \phi) &\propto p(v) \exp(\theta \cdot \phi_v) \\
&= \frac{\exp(\log p(v) + \theta \cdot \phi_v)}{\sum_{v=1}^{V} \exp(\log p(v) + \theta \cdot \phi_v)}
\end{aligned}
\tag{5.108}
$$

ここで $\phi_v \in \mathbb{R}$ は式(5.107)の Wordfish の場合と同様に, 単語 v の「極性」を

表すパラメータで, θ は 0 を中心とした標準正規分布

$$\theta \sim \mathcal{N}(0,1) \tag{5.109}$$

に従うとします.

式(5.108)より, このモデルでは Wordfish と同様に単語 v の確率は ϕ_v と θ の正負と大きさが一致すれば事前確率 $p(v)$ より高くなり, 逆になれば低くなる, というモデルになっています. $p(v)$ はコーパスから容易に計算できるため, このモデルは IRT の多項分布化, あるいは式(5.108)で表される多値ロジスティック回帰において, 説明変数 θ も回帰係数 ϕ も未知の場合の教師なし学習とみなすことができます.

このとき, テキスト d の確率は単語 v のテキスト内での頻度を n_{dv} とおくと

$$p(d|\theta,\phi) = \prod_{v=1}^{V} p(v|\theta,\phi)^{n_{dv}} \tag{5.110}$$

と書けますから, D 個のテキストからなるコーパス $\mathcal{D}$ 全体の確率は

$$p(\mathcal{D}|\Theta,\phi) = \prod_{d=1}^{D}\prod_{v=1}^{V} p(v|\theta_d,\phi)^{n_{dv}} \tag{5.111}$$

と表されます. Wordfish と同様に, 事前確率 (5.109) の下で式(5.111)を最大化するパラメータ $\Theta = \{\theta_1,\cdots,\theta_D\}$ および $\phi_1,\cdots,\phi_V$ を MCMC 法や EM アルゴリズムなどによって計算することができます.[*70]

ただし, こうするとたとえ単語 v と w が意味的に関係が深くても, ϕ_v と ϕ_w は別のパラメータとして推定しなければならないという問題があります. たとえば $\phi_{\text{good}} > 0$ と推定できても, これは $\phi_{\text{excellent}}$ や ϕ_{well} とは無関係で, excellent や well がコーパスに現れなければ, まったく学習することができません.

そこで, $\phi_1,\cdots,\phi_V$ を独立に学習するかわりに, 与えられたコーパスあるいは一般的なコーパスから事前に Word2Vec や GloVe などで学習された K 次元のニューラル単語ベクトル $\vec{v}$ を用いて, ϕ_v を

---

*70　これは多項分布によるモデルのため, ポアソン分布による Wordfish と異なり, パラメータが学習テキストの長さに依存せず, 新しいテキストについても適用することができるという特徴があります.

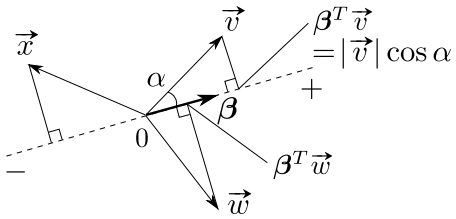


図 5.39: 方向ベクトル $\boldsymbol{\beta}$ による単語の極性の計算. $\boldsymbol{\beta}$ との内積, すなわち $\boldsymbol{\beta}$ 上への正射影の長さで単語の極性 $\phi_v = \boldsymbol{\beta}^T\vec{v}$ を計算します. ここで $|\boldsymbol{\beta}|=1$ で, $\vec{v}, \vec{w}, \vec{x}$ は単語ベクトルを表します. 単語ベクトルが $\boldsymbol{\beta}$ と逆方向の場合は, 極性はマイナスになります.

表 5.15: 表 5.4 の WRIME コーパスの極性語を参考に作成した, 標準的な極性辞書. $+$ が正例を, $-$ が負例を表します.

$+$	最高 嬉しい 楽しい 楽しみ 可愛い きれい しあわせ 好き かわいい
$-$	悪い 悲しい 寂しい 嫌い 怒り ない つらい 無理 しんどい めんどう

$$\phi_v = \boldsymbol{\beta}^T\vec{v} \qquad (\boldsymbol{\beta} = (\beta_1, \beta_2, \cdots, \beta_K)^T) \tag{5.112}$$

とモデル化します. この後で説明するように $\boldsymbol{\beta}$ の長さは 1 に正規化しますから, これは図 5.39 のように, 各単語ベクトル $\vec{v}$ と $\boldsymbol{\beta}$ のなす角を α としたとき, 単語の極性を $\vec{v}$ の $\boldsymbol{\beta}$ 上への正射影の長さ $\boldsymbol{\beta}^T\vec{v} = |\boldsymbol{\beta}||\vec{v}|\cos\alpha = |\vec{v}|\cos\alpha$ で「測って」いることに相当します. これにより, V 個の独立な $\phi_1, \cdots, \phi_V$ を求めるかわりに, K 次元の方向ベクトル $\boldsymbol{\beta}$ を一つだけ推定すればよいことになります. このとき, 式(5.108)は $\ell_v = \log p(v)$ とおけば,

$$p(v|\theta, \boldsymbol{\beta}) = \frac{\exp(\ell_v + \theta \cdot \boldsymbol{\beta}^T\vec{v})}{\sum_{v=1}^{V}\exp(\ell_v + \theta \cdot \boldsymbol{\beta}^T\vec{v})} \tag{5.113}$$

と表すことができます. 筆者が提案した[237]この方法は, 政治学分野で提案された, 301 ページの LSI をベースにしたベクトル空間におけるアルゴリズムである LSS (Latent Semantic Scaling) [238]の確率化ともみなすことができるため, 確率的 LSS (Probabilistic Latent Semantic Scaling, **PLSS**) とよぶことにします.

この $\boldsymbol{\beta}$ は, 単語埋め込みベクトルの空間において “良い–悪い”, “右翼–左翼” といった θ の極性を与える「極性軸」, あるいは「意味方向」を表しています.

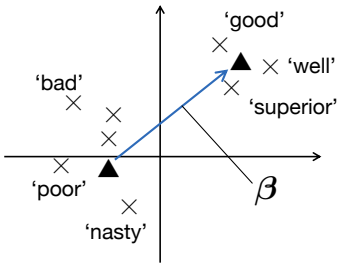


図 5.40: 極性辞書と単語ベクトルによる $\boldsymbol{\beta}$ の計算.

$\boldsymbol{\beta}$ は完全に教師なしで学習することもできますが, 316 ページ節で述べたように, そうして得られた $\boldsymbol{\beta}$ が分析の目的と一致しているとは限りません. そこで, PLSS では表 5.15 のように, 正例および負例として与える少数の極性語辞書から, 単語ベクトルを用いて $\boldsymbol{\beta}$ を次のように計算します. なお, $\boldsymbol{\beta}$ のノルムは 1 に正規化します.

$$\boldsymbol{\beta} \propto \left(\frac{1}{|S_+|} \sum_{v \in S_+} \overrightarrow{v} - \frac{1}{|S_-|} \sum_{w \in S_-} \overrightarrow{w} \right) \tag{5.114}$$

ここで S_+ は極性辞書のうち正の語の集合, S_- は負の語の集合です. 式(5.114)は単純に, $\boldsymbol{\beta}$ として負の単語ベクトルの平均から, 正の単語ベクトルの平均へ向かう方向をとることを表しています. この様子を図 5.40 に示しました. 極性辞書としては, 表 5.15 に示した標準的な極性辞書のように, ごく少数の単語のリストがあれば動きます. なお, 正例・負例の単語が「最も極端な語」である必要はなく, 意味方向のみが合っていればよいことに注意してください.

ここで「正例」「負例」とは必ずしも感情的な正負と関係していなくてもよく, 「右翼–左翼」「都会–田舎」「東洋–西洋」のように, 任意の意味的な軸を扱うことができます. ドイツの Schütze らは, 単語埋め込みの空間に実際にこうした, 与えられたタスクに関係する低次元の部分空間が存在することを発見し, これを**超密埋め込み** (Ultradense embedding) と呼んでいます[233]. 超密埋め込みは, 最も単純には, この場合のように 1 次元の部分空間となります. ただし, 予備実験でこの超密埋め込みを行列の固有ベクトルとして求める DensRay [239]

を使用したところ, 式(5.114)と比べて明らかにノイズの多い方向となったため, PLSS では単純な式(5.114)を採用することとしました.[*71]

表5.16 に, 表5.15 の標準的な極性辞書と, 後で説明する日本語 Wikipedia 記事から Word2Vec の方法で学習した $K=100$ 次元の単語ベクトルを用いて計算した単語の極性 $\phi_v = \boldsymbol{\beta}^T \vec{v}$ を示しました. 非常に少ない教師データにもかかわらず, 肯定的な単語および否定的な単語が, その強さとともに連続的に取り出せている様子がわかります.

極性辞書を用いる場合は, PLSS は単語ベクトルの計算以外にパラメータの学習を必要としません. 式(5.110)から, テキスト d と極性 θ の同時確率は

$$
\begin{aligned}
(5.115) \qquad p(d,\theta) &= \prod_{v=1}^{V} p(v\,|\,\theta,\boldsymbol{\beta})^{n_{dv}} \cdot p(\theta) \\
&= \prod_{v=1}^{V} \left(\frac{\exp\left(\ell_v + \theta \cdot \boldsymbol{\beta}^T \vec{v}\right)}{\sum_{v=1}^{V} \exp\left(\ell_v + \theta \cdot \boldsymbol{\beta}^T \vec{v}\right)} \right)^{n_{dv}} \cdot \mathcal{N}(\theta\,|\,0,1)
\end{aligned}
$$

となり, これを最大にするテキストの潜在的な極性 θ の MAP 解は, 1 次元の最適化で容易に計算することができます.

PLSS の実験 246 ページで使った WRIME コーパスのツイートの感情極性を, PLSS で分析してみましょう. 本書では基本的に Python を使用していますが, 政治学方法論分野では R が普及しているため, 本節では R で実装を行っています. 以下で使用しているスクリプトは, すべてサポートサイトからダウンロードすることができます.

情報が少ないと精密な分析ができないため, ここでは `wrime.test` の中で 20 単語以上のツイートを対象にすることにして, 次のようにしてテキストだけを抽出します. 結果は, 302 ツイートになりました. [*72]

---

*71 これは, DensRay の目的関数が, 極性辞書で単語 v の属する極性を $s(v)$ としたとき, 行列 $\mathbf{A} = \frac{1}{|S_+|}\sum_{s(v)=s(w)}^{(v,w):}(\vec{v}-\vec{w})(\vec{v}-\vec{w})^T - \frac{1}{|S_-|}\sum_{s(v)\neq s(w)}^{(v,w):}(\vec{v}-\vec{w})(\vec{v}-\vec{w})^T$ の固有ベクトルの計算に帰着され, 正例と負例の内部およびその間のペアを結ぶ個別のベクトルの和になっていることが原因だと考えられます. 式(5.114)と異なり, この定式化では S_+, S_- の中で和をとることでノイズを消す作用が働かず, v, w の選択に起因するノイズが $\mathbf{A}$ に残ってしまうからです.

*72 `cut` は, タブで区切られたテキストの指定された番号のフィールドを表示する, Unix の標準コマンドです. `awk 'NF>20'` とは, 空白で区切られたフィールド数が 20 より大きいとき標準ア

表 5.16: Wikipedia のコーパスから事前学習した単語ベクトルと, 表 5.15 の極性辞書を用いて計算した WRIME コーパスでの単語の極性 $\phi_v = \boldsymbol{\beta}^T \vec{v}$.

(a) 極性 $\phi_v > 0$ の上位語

ミッフィー	1.5639	星座	1.1436
チェキ	1.4430	スク	1.1274
キラキラ	1.3147	メゾン	1.1142
♪	1.2942	プレゼント	1.0828
かわいい	1.2609	リゾート	1.0742
アメニティ	1.2367	展	1.0505
クロワッサン	1.2283	土産	1.0492
たのしい	1.2268	ミント	1.0474
ミルキー	1.1964	マリオ	1.0255
しあわせ	1.1900	満喫	1.0134
ニベア	1.1663	美容	1.0097
水着	1.1637	hello	1.0028
応募	1.1579	ほっと	0.9945
アニメイト	1.1544	香菜	0.9894
♡	1.1466	コス	0.9867

(b) 極性 $\phi_v < 0$ の下位語

苦しん	−1.8143	許せ	−1.4058
耐え	−1.7748	察し	−1.3846
生じ	−1.7656	しまっ	−1.3837
治まっ	−1.7211	デメリット	−1.3749
予断	−1.6880	しまう	−1.3586
断ち切ら	−1.6177	起き	−1.3475
原因	−1.6022	断ち切っ	−1.3388
ざる	−1.5514	後悔	−1.3380
容赦	−1.5478	きれ	−1.3336
断ち切れ	−1.5090	症状	−1.3309
吐き	−1.4649	不安	−1.3211
ひどく	−1.4551	許さ	−1.3163
訴え	−1.4377	気付か	−1.3146
隙	−1.4274	村八分	−1.3140
困憊	−1.4211	雑言	−1.3083

```
% awk 'NF>20' wrime.test | cut -f 2 > wrime.txt
% head wrime.txt
先輩 の 息子 が 、 まじ 一瞬 だ けど つかまら ず 立つ ..
わたし の 伝え 方 が おかしい の か ? なぜ 、 全て や..
「 落ちる 」 場面 や 、 走っ て も 走っ て も 前 に 進..
Shift + win キー + S キー で 範囲 指定 の 画面 キャプ..
```

`wrime.txt` はこのように, 1 行に 1 文書 (1 ツイート) の単語が並んだ, 単なるテキストファイルです. PLSS は正解ラベルを必要としませんが, 検証のために取り出しておきましょう.

```
% awk 'NF>20' wrime.test | cut -f 1 > wrime.label
```

単語ベクトルの種類は任意ですが [*73], ここでは Wikipedia の記事から作成された単語ベクトルの公開データ Wikipedia2Vec [240] [*74] から, 100 次元の日本語単語ベクトルを使うことにします. サイトから “Japanese 100d(txt)” のベクトルを入手し, `bzip2` で解凍します.

```
% bzip2 -dc jawiki_20180420_100d.txt.bz2 > jawiki.vec
```

クション (その行を表示) を行う, という awk (73 ページ) のコマンドです.

*73　対象となるテキスト自体から, 3 章で紹介した方法で単語ベクトルを学習することも原理的には可能ですが, これには通常, かなり大量のテキストを必要とします.

*74　`https://wikipedia2vec.github.io/wikipedia2vec/`

分析に使う極性辞書は, 表 5.15 のものを用います. これは +,− の順に “ラベル<TAB>単語..” というフォーマットで, サポートサイトに posneg.ja という名前で置いてあります.

```
% cat posneg.ja
positive        最高 嬉しい 楽しい 楽しみ 可愛い きれ..
negative        悪い 悲しい 寂しい 嫌い 怒り ない つら..
```

この上で, 次のように plss を実行してツイートの極性を計算します. 使い方は plss の中を読むか, このスクリプトを単独で実行してみてください. 以下では R でテキストを扱うパッケージ quanteda[*75] および関連ライブラリがインストールされていることを前提にしていますので, エラーが出た場合は install.packages でインストールしておいてください.

```
    % plss wrime.txt posneg.ja jawiki.vec output
⇒   preparing word vectors..
    total 14129 words selected.
    running PLSS..
    loading wordvectors from /tmp/wordvec-217.vec.. done.
    preparing data.. done.
    documents = 8706, vocabulary = 14129
    computing theta..
    computing 8706/8706.. done.
    theta written to output.theta.
    phi   written to output.phi.
```

保存された output.theta が各ツイートの極性 $\theta \sim \mathcal{N}(0,1)$, output.phi は表 5.16 に示した, 内部で計算した単語の極性 ϕ_v です. (推定に用いなかった) 正解のラベルおよびツイート本文とともに表示してみましょう. 次のスクリプト theta.sh を実行すると, ランダムな 20 ツイートを極性 θ の値でソートして表示します.

```
    % theta.sh output.theta wrime.label wrime.txt
⇒   positive   1.8362  あー (*´∇`) キセキたちが私を萌殺そうとす..
    positive   0.9879  ぐっどこんでぃしょん。心も頭もクリア。秋分..
    positive   0.9121  やっぱ神席だった。キラーＴ細胞さんってマチ..
    positive   0.7518  ３週間ぶりに卓球してきました♪高校生の時の..
    positive   0.7212  ソフトバンク、楽天、元 zozo の前澤さんなど..
```

*75 http://quanteda.io/ ; 執筆時は quanteda 4.0 の上で実装しています.

```
negative   0.6797  あっ、ぶんぐ博の荷物忘れた…。今日、先生に..
positive   0.6390  高級牛乳買ってから寝る前に牛乳を 1 杯飲むよ..
positive   0.5202  今日発売のじょーちゃん載ってる雑誌さすがに..
positive   0.3314  同席のご夫婦さんに JAW さんファン歴何年ですか..
positive   0.0837  早速遊びすぎて電池無くなったよ!家の中って被..
positive  -0.0546  日々幸せだなぁって思ってるんだけど、今日ほん..
negative  -0.0575  あ、明日大阪に行ったら接触確認アプリの働きが..
negative  -0.2291  ほわ◯ぷワナビとか借◯玉ワナビ、大抵頭 (特に)..
positive  -0.4468  うおー肩と腰が痛いわーごみ出しに行かなきゃー..
negative  -0.6453  とある友人の言動が気になって仕方がない時があ..
negative  -0.6546  気にかけてくれる素晴らしい上司に恵まれて、な..
negative  -0.6990  どうしよう。来年行く旅行のことについて考えよ..
negative  -0.7387  うちの母親でもここまでやったことはないかなあ..
negative  -1.0035  体は疲労困憊なのに妙な緊張で眠れぬ今日もぐっ..
negative  -1.1121  4 時過ぎまで眠れず昨日打ち合わせた仕事モヤ..
```

このように, PLSS ではごく少数の極性語辞書だけで, テキストをその軸に沿って極性の強さに応じた連続値で並べることができ, 陽性–陰性の場合は人手で付与した極性とも, ほぼ一致していることがわかります.

極性辞書は陽性–陰性だけでなく, 任意の軸で作ることができます. 図 5.41 では, 右翼–左翼の政治的立場を表す (a) の極性語辞書を用いて, 5.1 節の livedoor コーパスの “topic-news” カテゴリーの記事を解析した様子を (b) に示しました. 記事にはもちろん, 右翼–左翼の軸に関するラベルはありませんが, 確かにこの極性軸に沿って文書を並べることができることがわかります.

5.6.2　PLSS の半教師あり学習

上では θ の極性を表す基準として LSS と同様に少量の極性辞書を用いましたが, こうした辞書が分析対象について自明に作成できるとは限りません. たとえば, 欧州において移民労働者についての賛成派と反対派を特徴づけるキーワードが, 分析前から明らかとは限らないからです.

しかし, そうした場合でも典型的な「正例」の文書と「負例」の文書は示せる場合が多いと考えられます. 直感的には, それぞれの文書に共通して現れる単語 (単語ベクトル) から, 間接的に単語の極性が導かれるはずです. コーパスのうち, こうした極性が既知の文書の集合を X_ℓ, それらへの 1/0 のラベルの集合を Y_ℓ とすると, 各文書とそのラベル $(y, d) \in (Y_\ell, X_\ell)$ について,

positive	保守 国家 伝統 経済 秩序 軍事 成長 資本
negative	平等 多様性 環境 平和 福祉 権利 労働

(a) 右翼–左翼の政治的立場を表す極性語辞書.

1.8022	「富士山をしろ」日本の軍事誌の内容に中国ネットユーザー..
1.6412	熊本の“政治家らしからぬ”経歴が話題 25 日に行われた熊..
1.4820	2011 年の日本の地震図に「すぎる」2011 年 1 月 1 日 00:00..
1.3375	富士山近郊でする地震に不安の声相次ぐ 1 月 29 日に発生し..
1.2284	中華圏で大ブレイクした“日本作り”19 日に「日経 NET」に..
1.1621	好きな女子アナランキングにネット騒然 9 日、STYLE が行っ..
1.1586	北朝鮮が´´ ミサイル´´ 発射も 1 分で落下 13 日、TBS を含..
1.1239	世界で有名な日本人ベスト 3 に意外な人物が続出 16 日放送..
:	:
−0.9924	猫ひろし、日本国籍再取得にも「当たり前だ」の声法律相談..
−1.0196	死刑廃止論者がネット掲示板でに 29 日、ダイヤモンドオン..
−1.0341	学校を休んでディズニーランドへは、ありかしかり、費、読..
−1.0834	生活保護受給をめぐり物議を醸す河本親子に法的措置の可能..
−1.1430	見知らぬ団体が勝手にハロプロの YouTube 公式チャンネルを..
−1.2003	東京都がまたマンガ・アニメ規制の動きか東京都が男女平等..
−1.2050	河本の姉が片山さつき氏に「後で謝ることになる」ネットで..
−1.4021	生活保護のイメージ悪化に抗議も、反論の声多数 30 日、生活..

(b) 解析した θ の上位および下位 8 個の記事.

図 5.41: PLSS による livedoor ニュースコーパスの解析. “topic-news” のラベルをもつ 770 個の記事の内容を, (a) の極性語辞書を用いて分析した結果を (b) に示しました. 右寄りな記事にはより高い θ が, 左寄りな記事にはより低い θ が推定されています.

$$p(y, d, \theta, \boldsymbol{\beta}) = p(y|\theta) \prod_{v=1}^{V} p(v|\theta, \boldsymbol{\beta})^{n_{dv}} \cdot p(\theta)p(\boldsymbol{\beta}) \tag{5.116}$$

を最大化する $\boldsymbol{\beta}$ を求めることを考えましょう. ここで第 1 項はロジスティック回帰

$$p(y=1|\theta) = \sigma(\theta) = \frac{1}{1+e^{-\theta}} \tag{5.117}$$

です. このグラフィカルモデルを図 5.42 に示しました. θ が大きい, あるいは小さい方が第 1 項の教師データに対する識別モデルの尤度が高くなりますが, 逆に第 2 項の単語の生成確率が下がる可能性があるため, 両者のトレードオフで θ が決まり, それによって式(5.115)から $\boldsymbol{\beta}$ が決まることになります.

ここで問題なのは, 回帰パラメータ $\boldsymbol{\beta}$ だけでなく, 文書の潜在的な極性 θ も未知なことです. θ および $\boldsymbol{\beta}$ を同時に最適化することも可能ですが, 心理統計学

においてこうした同時推定は一致性を持たないことが知られています[241]. また, 二値ラベルの y という弱い教師情報からは θ は一意には決まらず, θ を学習時に点推定することは, 教師データへの過学習をもたらす可能性があります.

適応的ガウス–エルミート求積による解法　そこで, θ を推定するかわりにモデルから積分消去し,

$$
\begin{aligned}
p(y, d, \boldsymbol{\beta}) &= \int_{-\infty}^{\infty} p(y, d, \theta, \boldsymbol{\beta}) d\theta \\
&= \int_{-\infty}^{\infty} p(y|\theta) \prod_{v=1}^{V} p(v|\theta, \boldsymbol{\beta})^{n_{dv}} p(\theta) d\theta \cdot p(\boldsymbol{\beta})
\end{aligned}
\tag{5.118}
$$

を $\boldsymbol{\beta}$ について最適化することを考えましょう [*76]. $p(\theta)$ は標準正規分布 $\mathcal{N}(0, 1)$ ですから, 式(5.118) の形のガウス分布に関する積分はガウス–エルミート求積 [*77] と呼ばれる方法で, きわめて正確に数値的に求めることができます.

ガウス–エルミート求積では, 関数空間での直交多項式を用いて, 図 5.43(a) に示したように関数 $f(x)$ の e^{-x^2} に関する積分を次の形で高精度に近似します.

$$
\int_{-\infty}^{\infty} f(x) e^{-x^2} dx \simeq \sum_{i=1}^{H} w_i f(x_i) \tag{5.119}
$$

ここで $\mathbf{x} = (x_1, \cdots, x_H)$ は分点 (abscissa) と呼ばれる座標, $\mathbf{w} = (w_1, \cdots, w_H)$ は対応する重みで, $H = 9$ のとき (本書の例では $H = 20$ としました) は

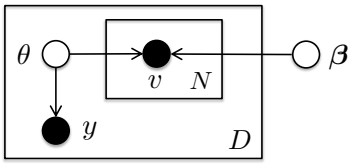
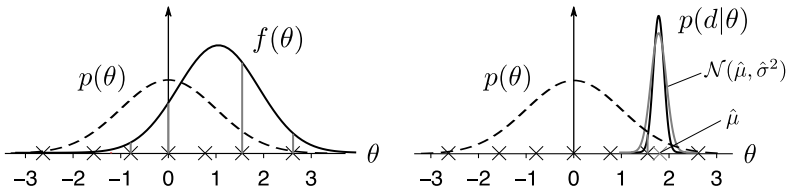


図 5.42: $\boldsymbol{\beta}$ を推定するための教師データのグラフィカルモデル. $\boldsymbol{\beta}$ だけでなく θ も未知のため, θ については周辺化を行って積分消去することで $\boldsymbol{\beta}$ を推定します.

---

*76　Hamiltonian MCMC 法[242]を用いた $\boldsymbol{\beta}$ のベイズ推定も検討しましたが, MAP 推定を用いる方が安定した結果となりました.

*77　求積 (quadrature) とは, 定積分の値を数値的に求めることです.



(a) 一般の関数 $f(\theta)$ の場合　(b) テキストの尤度 $p(d|\theta)$ の場合

図 5.43: 適応的なガウス–エルミート求積. 一般の関数 $f(\theta)$ の期待値 $\int f(\theta)p(\theta)d\theta$ は, (a) のように×印で示した分点 x_i での $f(\theta)$ と重み w_i から高精度に計算することができますが, テキストの場合は (b) のように尤度 $p(d|\theta)$ が一部の θ に集中するため, 変数変換を行って MAP 解 $\widehat{\mu}$ の周辺で適応的なガウス–エルミート求積を行い, 期待値 $\int p(d|\theta)p(\theta)d\theta$ を計算します.

$$\begin{aligned}\mathbf{x} &= (-3.191,\ -2.267,\ -1.469,\ -0.724,\ 0,\ 0.724,\ 1.469,\ 2.267,\ 3.191),\\ \mathbf{w} &= (0.00004,\ 0.005,\ 0.088,\ 0.433,\ 0.720,\ 0.433,\ 0.088,\ 0.005,\ 0.00004)\end{aligned}$$

です. これらの値は, 標準的なガウス–エルミート求積のパッケージで計算することができます.[*78] 式(5.119)は e^{-x^2} についての積分なので, $e^{-x^2}=e^{-\theta^2/2}$ すなわち $\theta=\sqrt{2}x$ とおけば, 変数変換により

$$\int_{-\infty}^{\infty} f(\theta)\mathcal{N}(\theta|0,1)d\theta = \int_{-\infty}^{\infty} f(\theta)\frac{1}{\sqrt{2\pi}}e^{-\frac{\theta^2}{2}}d\theta \simeq \frac{1}{\sqrt{\pi}}\sum_{i=1}^{H} w_i f(\sqrt{2}x_i) \tag{5.120}$$

と計算することができます.

ただし, 式(5.120)による積分は, そのままでは非常に効率が悪くなります. 一般にテキストは多くの単語を含むため, θ の事後分布はある値 $\widehat{\theta}$ の近くに集中しており, 式(5.120)ではほとんどの分点での尤度が 0 になってしまうからです. そこで, 統計学の分野で提案された[243]の方法を用いて, 積分を $\widehat{\theta}$ の周りで実行することにします. θ の事後分布を近似する平均 $\mu=\widehat{\theta}$ と分散 σ^2 は二分探索と二階差分により容易に求めることができますので, まず, $\phi=\mu+\sigma\theta$ と変数変換すると, 簡単な計算により

∗78 Python では `numpy.polynomial.hermite.hermgauss` で, R では `gaussquad` パッケージの `hermite.h.quadrature.rules` で計算できます.

$$\int_{-\infty}^{\infty} f(\theta)\mathcal{N}(\theta\,|\,\mu,\sigma^2)d\theta \simeq \frac{1}{\sqrt{\pi}}\sum_{i=1}^{H} w_i f(\mu+\sqrt{2}\sigma x_i) \tag{5.121}$$

であることがわかります. このとき, 求める積分を次のように変形します.

$$I=\int_{-\infty}^{\infty} f(\theta)\mathcal{N}(\theta\,|\,0,1)d\theta=\int_{-\infty}^{\infty}\underbrace{f(\theta)\,\frac{\mathcal{N}(\theta\,|\,0,1)}{\mathcal{N}(\theta\,|\,\mu,\sigma^2)}}_{h(\theta)}\mathcal{N}(\theta\,|\,\mu,\sigma^2)d\theta \tag{5.122}$$

式(5.122)の最初の 2 項を $h(\theta)$ とおけば, 式(5.121)より

$$I \simeq \frac{1}{\sqrt{\pi}}\sum_{i=1}^{H} w_i h(\mu+\sqrt{2}\sigma x_i) \tag{5.123}$$

と, 図 5.43(b) に示したように $\mathcal{N}(\theta\,|\,\mu,\sigma^2)$ についての適応的な積分で置き換えて求めることができます.

われわれの場合, 求めたい積分は式(5.118)でしたから, 対数で計算するために

$$\ell(\theta)=\log p(y\,|\,\theta)+\sum_{v=1}^{V} n_{dv}\log p(v\,|\,\theta,\boldsymbol{\beta}) \tag{5.124}$$

と定義すれば, $h(\theta)$ は

(5.125)

$$h(\theta)=e^{\ell(\theta)}\frac{\mathcal{N}(\theta\,|\,0,1)}{\mathcal{N}(\theta\,|\,\mu,\sigma^2)}=\exp(\ell(\theta)+\log\mathcal{N}(\theta\,|\,0,1)-\log\mathcal{N}(\theta\,|\,\mu,\sigma^2))$$

となります. 見やすくするために $y_i=\mu+\sqrt{2}\sigma x_i$ とおけば,

$$\begin{aligned} p(y,d,\boldsymbol{\beta}) &= \int_{-\infty}^{\infty} h(\theta)\mathcal{N}(\theta\,|\,\mu,\sigma^2)d\theta\cdot p(\boldsymbol{\beta}) \\ &\simeq \frac{\sigma}{\sqrt{\pi}}\sum_{i=1}^{H}\exp\left[\log w_i+\ell(y_i)+\frac{1}{2}\left(\frac{1}{\sigma^2}(y_i-\mu)^2-y_i^2\right)\right] \\ &\quad\cdot p(\boldsymbol{\beta}) \end{aligned} \tag{5.126}$$

となり, ℓ_i の中に $\boldsymbol{\beta}$ が含まれるこの式の対数を $\boldsymbol{\beta}$ について偏微分し, L-BFGS 法で最適化することで $\boldsymbol{\beta}$ を計算します. 式(5.126)の対数の偏微分は, 計算する

と文書 d の長さを L として,

$$\frac{\partial}{\partial \boldsymbol{\beta}} \log p(y, d, \boldsymbol{\beta}) = L\theta \left[\frac{1}{L} \sum_{v \in d} \vec{v} - \sum_{v=1}^{V} p(v \mid \theta, \boldsymbol{\beta}) \vec{v} \right] \tag{5.127}$$

という直感的にも妥当な結果が得られます.

PLSS の実験 (続き) それでは, 実際のテキストを分析してみましょう. ここでは, 国会の議事録を使ってみることにします. 国会会議録検索システム [*79] から, 2023 年度の第 211 回通常国会での衆議院・文部科学委員会のテキストをダウンロードしてみましょう. これは 16 回の開催で, サポートサイトにあるスクリプトを用いて次のように実行すると, 2,210 個の発言が `livedoor.txt` と同じ「発言者<TAB>発言内容..」の形式で得られます.

```
% diet-split.py *.txt | diet-format.py > all.txt
```

このうち, 次のように実行して大臣および委員長の司会・答弁および 50 文字未満の短い発言を除外すると, 984 個の単語分割された発言とその発言者が得られました. [*80]

```
% diet-sieve.awk all.txt | cut -f 2 | mecab -O wakati \
  -b 65536 > education.txt
% diet-sieve.awk all.txt | cut -f 1 > education.label
```

発言の多い上位順に宮本岳志委員 (日本共産党), 柚木(ゆのき)道義委員 (立憲民主党), 西岡秀子委員 (国民民主党) のうち, 違いの自明でない後者二名の発言から次のようにしてランダムに 20 個の発言を抽出し, これを正例–負例とみて違いの軸を抽出してみることにしましょう. この二者はいずれも野党です.

```
% article-id.py education.label 柚木 西岡 20 > yn.id
% cat yn.id
柚木    4,6,8,11,152,154,157,165,167,231,232,235,236,242, ..
西岡    100,103,106,109,279,281,285,401,511,513,519,745, ..
```

このファイルのフォーマットは, 名前<TAB>番号 1, 番号 2.. で, 文書番号は 1 から数え, 1 行目が正例に, 2 行目が負例に対応します. この上で, 次のようにし

*79 `https://kokkai.ndl.go.jp/`

*80 この場合のように 1 行が長いと MeCab の解析バッファが足りなくなるため, `-b 65536` のようにしてバッファを標準の 4KB から 64KB に増やしています.

て PLSS の半教師あり学習を実行します. 内部で plss-semi.R を呼んでいますので, 先にこのスクリプトを単体で実行して, 必要なライブラリをインストールしておいてください.

```
% plss-semi education.txt yn.id jawiki.vec output
preparing word vectors..
total 8031 words selected.
running PLSS-semi..
loading wordvectors from /tmp/wordvec-950.vec.. done.
preparing data of language ‘‘ja’’.. done.
optimizing beta from examples..
iter    1 value 15.901530
iter    2 value 14.383028
iter    3 value 12.641068
:
iter   33 value 10.205899
final  value 10.205899
converged
number of docs = 984
word dimension = 100
norm of beta   = 2.24
computing theta..
computing 984/984.. done.
theta written to output.theta.
phi   written to output.phi.
```

output.theta および output.phi が推定した各発言の極性 θ と, 学習した $\boldsymbol{\beta}$ に沿った単語の極性 $\phi_v = \boldsymbol{\beta}^T \vec{v}$ です. 次のように実行すると, θ および発言者, 発言内容を並べて表示し, θ でソートすることができます.

```
% paste output.theta education.label education.txt \
  | sed 's/ //g' | sort -nr
⇒  1.400558 柚木委員 簗副大臣、お考えについては答えてください。L..
 1.369929 柚木委員 そんな、文部科学副大臣として、今おっしゃって..
 1.292844 柚木委員 ということは、永岡文部科学大臣としては、L G..
 :
-1.292683 西岡委員 大変様々な要件が課されておりまして、また今後..
-1.305099 西岡委員 これまでも議論になっておりますけれども、やは..
-1.333402 西岡委員 関連いたしまして、先ほども申し上げましたよう..
```

θ	発言者	発言内容
0.9843	柚木委員	まとめて答えていただいたので、最後、ちょっとだけ時間ができたので。..
0.7876	柚木委員	ちょっと警察庁、この後、答弁いただきます。　伊佐副大臣、最後の質問..
0.7387	梅谷委員	そうですね。　じゃ、この組織的あっせん事件の中心的人物が今大学で役..
0.7260	柚木委員	これは驚きの答弁ですよ、大臣。　私、一応、ＬＧＢＴは種の保存に背く..
0.6559	宮本委員	学校給食法第十一条は障害にならない、当たり前です。　しかも、自民党..
0.3464	吉川委員	指針ですけれども、先ほど触れたとおり、強制力を伴わない、望ましいだ..
0.3308	柚木委員	もうこの項目は最後にしたいと思いますが、だとすると、まさに十三ペー..
0.2456	柚木委員	非常に、今後の方向性をある意味中間報告的に今整理して御答弁いただい..
0.1865	梅谷委員	検討中という話ですので、これ以上は申し上げませんが。　ただ、私がも..
−0.0026	菊田委員	大切なことは、やはり、現場の教職員の先生方がすごいプレッシャーとか..
−0.0574	宮本委員	幼児教育無償化したからやれという話じゃなくて、されなくてももちろん..
−0.1607	宮本委員	おっしゃるとおりで、経緯があって、だから多様になっているということ..
−0.2457	吉川委員	これから検討し、またパブコメ等ということですけれども、私は、審議会..
−0.2807	牧委員	確かにおっしゃるように、ガバナンスは大切なんですけれども、逆に、萎..
−0.3307	宮本委員	当然ですね。　さらに、私に対応してくださった金沢大学の職員のお一人
−0.4056	鰐淵委員	ありがとうございます。　今御紹介もありましたけれども、当初は評議員..
−0.4151	早坂委員	先ほど、私も私学出身で、両親が共働きで、どうにか学校を出していただ..
−0.5283	宮本委員	教育機会確保法附則の三には、「この法律の施行後三年以内にこの法律の..
−0.6099	西岡委員	基準、要件については、これから法案が成立した後、審議会で議論をする..
−1.0058	西岡委員	やはり、支出の妥当性、透明性は大変重要だと思いますので、しっかりそ..

(a) 各発言に推定された θ (一部)

勝目	3.0700	簗	2.3292	ワーキング..	−2.6780	学寮	−2.0325
辺野古	3.0331	浮島	2.3156	産学	−2.3894	防災	−2.0261
ワールドカ..	2.6581	恥ずかしい	2.2712	気象	−2.2831	周期	−2.0104
利府	2.6491	飲酒	2.2636	か年	−2.2318	高度	−2.0074
有権者	2.5233	わいせつ	2.2398	凝らし	−2.2294	豊橋技術科学..	−1.9977
中体連	2.4912	英弘	2.2019	耐震	−2.2254	科目	−1.9972
左折	2.4738	下線	2.1374	災	−2.2171	カリキュラム	−1.9865
衆議院	2.4640	起点	2.1102	整い	−2.1866	不断	−1.9793
双葉	2.4262	知事	2.0699	屋根	−2.1721	地震	−1.9738
乗車	2.3836	野球	2.0688	くらし	−2.1421	築い	−1.9514
$\phi_v > 0$ (柚木議員側)				$\phi_v < 0$ (西岡議員側)			

(b) 推定された単語の極性

図 5.44: PLSS の半教師あり学習による極性の推定. 極性辞書を使わなくても, 与えた文書集合のうち, 少数の正例と負例の文書の番号を与えれば, 極性を表す軸 $\boldsymbol{\beta}$ をそこから学習することができます.

この全体からランダムに抽出した 20 発言とその極性を図 5.44(a) に示しました. 現場感覚を大事にするアクティブな柚木議員に対し, 文教族の参議院議員を父親に持つ西岡議員は, より制度的なアプローチをとっていることが読み取れます. この違いは, ϕ_v にも見てとることができます. 図 5.44(b) では確かに, 柚木議員 (+ 側) では「左折」「乗車」「野球」といった単語の極性値が高く, 西岡議員 (− 側) では「産学」「耐震」「防災」といった単語の極性値が高くなってお

り, 同じ野党でありながらも, 興味の違いや政治姿勢を反映していることがわかります. また他の議員の立場も同じ軸の上で解釈することができ, 全体的な構造が一見不明な文部科学委員会での議論に, 一定の計量的な視点を与えることができました.

5 章のまとめ

5 章では, 文書の意味的内容を数学的に表現する文書モデルについて学びました. ナイーブベイズ法によって文書を確率的に, かつ高速に分類することができます. 教師なしナイーブベイズ法であるユニグラム混合モデル (UM) は文書を確率的にクラスタリングするもので, EM アルゴリズムを使って学習することができます. Gibbs サンプリングによってベイズ学習することも可能で, 教師なしで興味深い潜在トピックが得られます. UM でさらに単語のバースト性も考慮したのが, バイオインフォマティクス分野でも使われているディリクレ混合モデル (DM) でした. 文書が複数のトピックからなると考えられる場合は, 潜在ディリクレ配分法 (LDA) がより柔軟な統計モデルで, 一般に「トピックモデル」として知られているのはこの LDA です. LDA も Gibbs サンプリングで効率的に学習することができ, 解釈性が高いために多くの応用があります.

上とは異なり, 文書の内容を実数値のニューラル文書ベクトルで表すことで, 単語ベクトルと同様に文書の意味をベクトル空間でとらえられるようになります. 3 章で学んだ Word2Vec を拡張することで, 同様にして自己相互情報量 (PMI) の行列分解で最適解を計算することが可能で, 本書ではこれを DocVec とよびました. 3 章で使った PMI およびその拡張である NPMI (正規化 PMI) は, トピックモデルの解釈にもたいへん有効です.

本書の最後に, 筆者の研究でやや高度ですが, 文書ベクトルのなす潜在空間にある観点での正–負の「極性軸」を考え, 少数の例から極性軸の方向を学習することのできる確率的潜在意味スケーリング (PLSS) について紹介しました. PLSS では, 心理統計学における項目反応理論の考え方を使うことで, 単純な分類と異なり, 文書にこの軸上で 0 を中心とした連続的な極性値を与えることができ, さまざまな方向で文書を分析することができます.

5 章の演習問題

[5–1] ナイーブベイズ法の計算で式(5.9)の平滑化パラメータ α の値を変えると, 性能にどのような影響があるでしょうか. α を単語ごとに変えることにし, 114 ページの方法で $\boldsymbol{\alpha}$ を最適化すると, 性能はどう変わるでしょうか.

[5–2] ナイーブベイズ法で, 単語のカテゴリ所属確率 $p(y|w)$ が特定のカテゴリについて高い語には, どんなものがあるでしょうか. $p(y|w)$ がどのくらい特定の y に集中しているかは, 確率分布のエントロピー (式(2.66)) で測ることができます. $p(y|w)$ のエントロピーの大小で単語をソートすると, どんな結果になるでしょうか.

[5–3] livedoor コーパスの各カテゴリでの単語分布 $p(w|y)$ から, NPMI を使って, 各カテゴリの特徴語を計算してみましょう. 上位語, 下位語はどのようになっているでしょうか. また, UM で教師なしで得られたカテゴリの特徴語とは, どのような違いがあるでしょうか.

[5–4] ユニグラム混合モデルについても, 学習したモデルから各単語 w のトピック事後確率 $p(z|w)$ を計算してみましょう. 意外なトピックに事後確率が高い単語はあるでしょうか.

[5–5] 任意の単語 w_1, w_2, w_3 を 3 つ選んで, 語彙がこれらの単語だけからなるとしたとき, 適当なコーパスの各文書について最尤推定で計算した確率分布 $\mathbf{p}=(p_1, p_2, p_3)$ を, 図 5.14 のように単体上にプロットしてみましょう. 意味的に相関のある単語のペアを選んだとき, プロットにはどんな傾向が現れるでしょうか.

[5–6] DM で学習されたディリクレ分布のハイパーパラメータ $\boldsymbol{\alpha}_k$ を式(3.30)のように正規化してトピック k の多項分布 $\mathbf{p}$ の期待値を求めることで, 各トピックがどんな意味を持っているかを調べてみましょう. UM と何か違いはあるでしょうか.

[5–7] 5.3 節で行ったように, 適当な小説などのテキストに対して, ある単語が出現してから次に出現するまでの間隔 (何単語後か) をプロットしてみましょう. 地震のようなイベントがランダムな時間に起きる様子を記述する

ポアソン過程の言葉では, これは前方再起時間とよばれています. 単語によって, どんな特徴があるでしょうか. この分布をモデル化するには, どうするとよいでしょうか. (ヒント: 270 ページ脚注)

[5–8] 単語の長さ, あるいは 4 章 (190 ページ) で調べた文の長さの分布は, 文書の内容によって異なるのではないかと考えられます. UM や DM によるトピック別に分けて調べると, どのような違いがあるでしょうか. また, 文書の内容を表すトピック分布や文書ベクトルから, 分布のパラメータを予測する回帰モデルを学習することはできるでしょうか.

[5–9] 文書の特徴は, 本章のように単語の頻度を使うものに限りません. 文字または文字種 (漢字, ひらがな, カタカナ, 記号など) の頻度で文書モデルを作ると, どんな知見が得られるでしょうか.

[5–10] livedoor コーパスについて LDA を使って学習した各文書のトピック分布 $\boldsymbol{\theta}$ と教師ラベル y は, どのように対応しているでしょうか. ラベル y ごとに, どんなトピックが多いのかを集計してみましょう. また, 教師ラベルとしては, どんな話題が足りなかったといえるでしょうか.

[5–11] 適当なコーパスで計算した文書ベクトルに 5.5.2 節のように ICA を適用し, どんな意味的な軸が得られるかを調べてみましょう. 特に, 表 5.13 と表 5.14 のように, 軸の値の大きい方と小さい方にどのような対応関係があるでしょうか. ICA では軸の正負自体に意味はなく, 反転しても同じモデルであることに注意してください.

[5–12] 3 章で前後の共起窓から求めた単語ベクトルについても, 同様に ICA で変換した独立成分軸を求めてみましょう. 本章で文書内全部を共起の対象とした単語ベクトルと, 何か違いがあるでしょうか. また, Word2Vec, GloVe, SVD による単語ベクトルの計算法で何か差がみられるでしょうか.

[5–13] 同じコーパスでトピックモデルと文書ベクトルを学習したとき, 学習された表 5.9 のようなトピックと, 表 5.13 のような ICA の独立成分軸はどのように異なるでしょうか. どのトピックと軸が似ており, どれが似ていないかを定量化するには, どうすればいいでしょうか.

[5–14] PLSS で, 表 5.15 に示したような positive–negative 以外の辞書を使って

テキストを分析してみましょう. こうした極性語辞書を, 与えられた ± のラベルの文書から自動的に作るには, どうするといいでしょうか. (ヒント: NPMI)

5 章の文献案内

本章で使った Gibbs サンプリング (MCMC 法の一種) のようなベイズ的な方法を体系的に学ぶには, 2 章の文献案内で紹介した PML や PRML といった機械学習の教科書だけでなく, 最近翻訳の出た Hoff の『標準ベイズ統計学』[244] (原著:[245]) や, Gelman らの『ベイズデータ解析 (第 3 版)』[170] (原著:[246], 通称 BDA3) のようなベイズ統計学の教科書を読むとよいでしょう. 前者は要点がコンパクトにまとまっており, 版を重ねて大部になった BDA3 よりも入門として勧められます. ベイズに限らない標準的な統計学の教科書としては, 筆者は Casella と Berger の “Statistical Inference” [247]を主に参照しています. ベイズ統計学を学ぶために筆者が最初に読んだのは, Lee の “Bayesian Statistics: An Introduction” [248]でした. これはやや数学的ですが, 定評のある入門書で, 「初めての都市を訪れて, 見た最初のバスの番号が 17 番だった. このとき, この都市には何番までのバスがあると考えられるか?」といった, ベイズ統計でないと解けない, 興味深い問題についても議論されています.

ベイズ学習以前によく使われていた EM アルゴリズムについては, 上記の教科書でも解説されていますが, Neal と Hinton による[249]は見通しのよい説明で, 大変参考になるものです. EM アルゴリズムおよび変分ベイズ EM アルゴリズムは, VAE [250]のような現代の深層学習の方法の基礎にもなっています.

本章で学んだ UM, DM, LDA といったトピックモデルは, 岩田による『トピックモデル』[251]でも (本書よりやや簡潔に) 紹介されています. 詳しく学ぶには, 佐藤による[84]が最も専門的な話題を扱っています. なお, トピックモデルについては, LDA の原論文[185]は 5.4.3 節で説明したような重要な論点を含んでおり, 一読をお薦めします. LDA の学習は最初に変分ベイズ法によるものが提案され, その後に心理学者の Griffiths によって, 本書に示したような Gibbs サンプリングによる解法が示されました[201]. LDA に関する多くの資料と異なり, この導出は Γ 関数などを使わない簡明なものです.

社会科学においてもテキストデータの利用は急速に広がっており, 多くの場合で単純にラベルを予測する教師あり学習とはみなせないことから, 本書で示したような方法が必要になっています. 政治学方法論では今井による “Quantitative

Social Science: An Introduction” [252] (和訳:『社会科学のためのデータ分析入門』[253]) の 5.1 節が, テキストデータ解析の入門にあてられています. 2022 年刊の “Text as Data: A New Framework for Machine Learning and the Social Sciences” [254]は社会科学におけるテキストデータ解析の専門書です. 経済学では, 別の著者による同じタイトルの “Text as Data” [255]が広く読まれているほか, 2023 年の “Text Algorithms in Economics” [256] で, 経済学におけるテキストの最新の使用法がまとめられています. 論文は多数ありますが, たとえば自然言語処理のトップ国際会議 ACL で 2015 年に発表された[257]は, アメリカ議会におけるティーパーティー運動の役割を階層的な項目反応理論 (318 ページ) および, 演説および法案のテキストに対する LDA を組み合わせて分析した, 非常に高度なモデルで興味深いものです.

あとがきと謝辞

本書の執筆依頼を受けたのは, 筆者がまだ京都の NTT コミュニケーション科学基礎研究所にポスドク研究員として在籍していた, 2008 年のことでした. 『数学セミナー』2007 年 11 月号に書いた記事「生きた言葉をモデル化する」[258] がきっかけだったと記憶しています. 何と 17 年もの長い間, 諦めずに待っていただいた岩波書店編集部の皆様, 編者の皆様, そして本書の刊行を期待していただいていた読者の皆様に, まずは最大の感謝を申し上げたいと思います.

この間に自然言語処理は深層学習時代を迎え, 筆者も統計数理研究所に移りました. 学生や企業の方々とさまざまな共同研究を行う中で, 機械学習や統計, 情報理論の基礎が忘れられていること, および文系も含む広い範囲の方が読める, わかりやすい教科書の必要性を感じたことが, 執筆を再開する最大の動機になりました. 内容としては, 依頼当初は本書の 5 章にあたる内容を考えていましたが, より基礎的な 2 章〜 4 章も含めて体系的な教科書にできたのは, この長い期間に, 筆者の経験や考察を深められたお蔭だと思っています.

本書は, 前著である『ガウス過程と機械学習』[12]が 2019 年に出版された後の 2020 年秋ごろに執筆を再開し, 他の仕事の合間を縫いつつ, 実装や計算例も含めて 4 年間の厖大な時間をかけて執筆しました. 今は, それだけの意味のある本になっていることを願うばかりです.

草稿をサポートページで公開した後, 読んでいただいた多くの方々から, 内容に関する貴重なコメントをいただきました. 近藤泰弘氏 (青山学院大), 松尾晃孝氏 (エセックス大), 西川賢氏 (津田塾大), 谷口忠大氏 (京大), 深谷覚氏 (産総研), 大羽成征氏 (ミイダス), 和田匡路氏 (東北大), 福元健太郎氏 (東大), 江島舟星氏 (Netflix), 山田武士氏 (近大) の皆様, および一部の一般の皆様で, 特に松浦健太

郎氏 (ホクソエム) からは本書の初期の段階より, 詳細なコメントをいただきました. また渡辺耕平氏 (ラザード・アセット・マネジメント) には, 政治学方法論分野でのテキスト分析について様々に教えていただきました. この場をお借りして, 皆様に感謝を申し上げたいと思います. 執筆が長期にわたったため, 岩波書店の歴代の担当編集者の吉田宇一様, 首藤英児様, 濱門麻美子様, 彦田孝輔様には, 様々な面でたいへんお世話になりました. 本書を読まれた皆様が, 言語は面白い, 統計は面白い, と感じていただけることを願っています.

筆者は小さな頃から, 言葉が非常に得意な子供でした. 言葉に対する興味が, 数学と計算機を通じて本書のような形で結晶化したと言えそうです. 長期間の執筆を支えていただいた家族に感謝するとともに, これまでにお世話になった諸先生方に, 本書を捧げたいと思います.

2024 年秋　東京にて　持橋 大地

付録

A　ディリクレ分布の積分と期待値

A.1　ディリクレ分布の積分公式

式(3.33)のディリクレ分布の正規化項に現れる積分公式 (式(3.38))

$$\int_0^1 \cdots \int_0^1 p_1^{\alpha_1-1} \cdots p_K^{\alpha_K-1} dp_1 \cdots dp_{K-1} = \frac{\Gamma(\alpha_1)\cdots\Gamma(\alpha_K)}{\Gamma(\alpha_1+\cdots+\alpha_K)} \tag{A.1}$$
$$(p_K = 1 - p_1 - \cdots - p_{K-1})$$

すなわち,

$$\Gamma(\alpha_1)\cdots\Gamma(\alpha_K) = \Gamma(\alpha_1+\cdots+\alpha_K) \int_0^1 \cdots \int_0^1 p_1^{\alpha_1-1} \cdots p_K^{\alpha_K-1} dp_1 \cdots dp_{K-1} \tag{A.2}$$

は, 次のようにして示すことができます.

式(3.34)のガンマ関数の定義から, 式(A.2)の左辺は

$$\begin{aligned}\Gamma(\alpha_1)\cdots\Gamma(\alpha_K) &= \int_0^\infty e^{-t_1} t_1^{\alpha_1-1} dt_1 \cdots \int_0^\infty e^{-t_K} t_K^{\alpha_K-1} dt_K \\ &= \int_0^\infty \cdots \int_0^\infty e^{-(t_1+\cdots+t_K)} t_1^{\alpha_1-1} \cdots t_K^{\alpha_K-1} dt_1 \cdots dt_K\end{aligned} \tag{A.3}$$

と表すことができます.

ここで, $u_1 + \cdots + u_K = 1$ $(u_k \geq 0)$ について $t_1 = u_1 y, \cdots, t_K = u_K y$ と変数変換すれば, $\dfrac{dt_1}{du_1} = \cdots = \dfrac{dt_K}{du_K} = y$ ですから, 式(A.3)の右辺は

(A.4)

$$\begin{aligned}
&\int_0^\infty \int_0^1 \cdots \int_0^1 e^{-(u_1 y + \cdots + u_K y)} (u_1 y)^{\alpha_1 - 1} \cdots (u_K y)^{\alpha_K - 1} dy \, (y du_1) \cdots (y du_{K-1}) \\
&\quad = \int_0^\infty \int_0^1 \cdots \int_0^1 e^{-(u_1 + \cdots + u_K) y} \, y^{\alpha_1 + \cdots + \alpha_K - 1} \, u_1^{\alpha - 1} \cdots u_K^{\alpha_K - 1} dy du_1 \cdots du_{K-1} \\
&\quad = \int_0^\infty e^{-y} \, y^{\alpha_1 + \cdots + \alpha_K - 1} dy \; \times \int_0^1 \cdots \int_0^1 u_1^{\alpha - 1} \cdots u_K^{\alpha_K - 1} du_1 \cdots du_{K-1} \\
&\quad = \Gamma(\alpha_1 + \cdots + \alpha_K) \int_0^1 \cdots \int_0^1 p_1^{\alpha - 1} \cdots p_K^{\alpha_K - 1} dp_1 \cdots dp_{K-1}
\end{aligned}$$

となり, 式(A.2)が証明できました. □

A. 2 ディリクレ分布の期待値

公式(A.1)を使うと, ディリクレ分布の期待値の公式 (式(3. 30)) も容易に示すことができます. ディリクレ分布 $\mathrm{Dir}(\boldsymbol{\alpha})$ に従う多項分布 $\mathbf{p} = (p_1, \cdots, p_K)$ について, k 番目の要素 p_k の期待値は, ディリクレ分布の式(3. 33)から

(A.5)
$$\begin{aligned}
\mathbb{E}[p_k] &= \int_0^1 \cdots \int_0^1 p_k \cdot \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^K p_k^{\alpha_k - 1} \, dp_1 \cdots dp_{K-1} \\
&= \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \int_0^1 \cdots \int_0^1 \prod_{k=1}^K p_k^{\alpha'_k - 1} \, dp_1 \cdots dp_{K-1}
\end{aligned}$$

と書くことができます. ただし $\alpha'_j = \alpha_j \ (j \neq k)$, $\alpha'_k = \alpha_k + 1$ です. よって, 式(A.5)の右辺は, 公式(A.1)から, $\alpha_1 + \cdots + \alpha_K = \alpha$ とおくと

(A.6)
$$\begin{aligned}
&\frac{\Gamma(\alpha)}{\prod_k \Gamma(\alpha_k)} \cdot \frac{\Gamma(\alpha_1) \cdots \Gamma(\alpha_k + 1) \cdots \Gamma(\alpha_K)}{\Gamma(\alpha_1 + \cdots + \alpha_K + 1)} \\
&= \frac{\Gamma(\alpha)}{\Gamma(\alpha + 1)} \cdot \frac{\Gamma(\alpha_k + 1)}{\Gamma(\alpha_k)} = \frac{\cancel{\Gamma(\alpha)}}{\alpha \cancel{\Gamma(\alpha)}} \cdot \frac{\alpha_k \cancel{\Gamma(\alpha_k)}}{\cancel{\Gamma(\alpha_k)}} = \frac{\alpha_k}{\alpha}
\end{aligned}$$

となります. ここで 2 行目では, 公式(3. 35)を用いました. よって, 公式(3. 30)が示されました. □

B　ディリクレ分布の $\boldsymbol{\alpha}$ のベイズ推定

ポリア分布の式

$$p(\mathbf{n}|\boldsymbol{\alpha}) = \underbrace{\frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N+\sum_k \alpha_k)}}_{(\diamondsuit)} \underbrace{\prod_{k=1}^{K} \frac{\Gamma(\alpha_k+n_k)}{\Gamma(\alpha_k)}}_{(\spadesuit)} \tag{B.1}$$

は α_k がガンマ関数 $\Gamma()$ の中に入っているため, そのままではサンプリングできる形になりません. しかし, 巧妙な補助変数 θ, x を導入すると [*1], ガンマ事後分布からサンプリングすることができます[59, Appendix C].

($\diamondsuit$)　ディリクレ分布で 2 次元の場合を考えると, **ベータ関数**

$$B(a,b) = \int_0^1 \theta^{a-1}(1-\theta)^{b-1} d\theta = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{B.2}$$

が得られます. よって,

$$B(\textstyle\sum_k \alpha_k, N) = \dfrac{\Gamma(\sum_k \alpha_k)\Gamma(N)}{\Gamma(\sum_k \alpha_k + N)} \tag{B.3}$$

ですから, ($\diamondsuit$) は補助変数 θ を積分消去した形で,

$$\frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N+\sum_k \alpha_k)} = \frac{B(\sum_k \alpha_k, N)}{\Gamma(N)} = \frac{1}{\Gamma(N)} \int_0^1 \theta^{\sum_k \alpha_k - 1}(1-\theta)^{N-1} d\theta \tag{B.4}$$

と表すことができます.

($\spadesuit$)　119 ページのコラムでみたように, Pochhammer 関数 $\Gamma(\alpha+n)/\Gamma(\alpha)$ は

$$\frac{\Gamma(\alpha+n)}{\Gamma(\alpha)} = \alpha(\alpha+1)\cdots(\alpha+n-1) = \prod_{i=0}^{n-1} (\alpha+i) \tag{B.5}$$

と表すことができますが, これも $\{0,1\}$ の補助変数 x_{ki} を周辺化した形で

$$\frac{\Gamma(\alpha_k+n_k)}{\Gamma(\alpha_k)} = \prod_{i=0}^{n_k-1} (\alpha_k+i) = \prod_{i=0}^{n_k-1} \sum_{x_{ki}\in\{0,1\}} \alpha_k^{x_{ki}} i^{1-x_{ki}} \tag{B.6}$$

---

*1　これは MCMC 法の基本テクニックの一つで, **補助変数法**とよばれています[246].

と書くことができることに注意しましょう.

よって, α_k がガンマ事前分布

$$p(\alpha_k) = \mathrm{Ga}(a,b) = \frac{b^a}{\Gamma(a)}\alpha_k^{a-1}e^{-b\alpha_k} \tag{B.7}$$

に従っているとすると, 式(B.4), 式(B.6)より, α_k の事後分布は, ベイズの定理から

$$p(\boldsymbol{\alpha}|\mathbf{n}) \propto p(\mathbf{n}|\boldsymbol{\alpha})p(\boldsymbol{\alpha}) = \prod_{k=1}^{K}\frac{b^a}{\Gamma(a)}\alpha_k^{a-1}e^{-b\alpha_k}\times \frac{1}{\Gamma(N)}\int_0^1 \theta^{\sum_k \alpha_k-1}(1-\theta)^{N-1}d\theta \times \prod_{k=1}^{K}\left(\prod_{i=0}^{n_k-1}\sum_{x_{ki}\in\{0,1\}}\alpha_k^{x_{ki}}i^{1-x_{ki}}\right) \tag{B.8}$$

となります. これは θ と x_{ki} を周辺化した形ですから, 同時確率は

$$p(\boldsymbol{\alpha},\theta,x|\mathbf{n}) \propto \prod_{k=1}^{K}\frac{b^a}{\Gamma(a)}\alpha_k^{a-1}e^{-b\alpha_k} \times \frac{1}{\Gamma(N)}\theta^{\sum_k \alpha_k-1}(1-\theta)^{N-1} \times \prod_{k=1}^{K}\left(\prod_{i=0}^{n_k-1}\alpha_k^{x_{ki}}i^{1-x_{ki}}\right) \tag{B.9}$$

です. よって各 α_k の事後分布は, 式(B.9)から α_k に関する項を抜き出せば, 補助変数 θ, x_{ki} が与えられた下では

$$\begin{aligned} p(\alpha_k|\mathbf{n},\theta,x) \propto p(\alpha_k,\theta,x|\mathbf{n}) &= \alpha_k^{a-1}\cdot e^{-b\alpha_k}\cdot\theta^{\alpha_k}\cdot\prod_{i=0}^{n_k-1}\alpha_k^{x_{ki}} \\ &= \alpha_k^{a+\sum_{i=0}^{n_k-1}x_{ki}-1}e^{-(b-\log\theta)\alpha_k} \\ &\sim \mathrm{Ga}(a+\textstyle\sum_{i=0}^{n_k-1}x_{ki},\, b-\log\theta) \end{aligned} \tag{B.10}$$

となることがわかります. 同様にして補助変数 θ, x_{ki} についても対応する項を抜き出すと, その分布は

$$\begin{aligned} p(\theta|\mathbf{n},\boldsymbol{\alpha},y) &\sim \mathrm{Be}(\textstyle\sum_k\alpha_k, N) \\ p(x_{ki}|\mathbf{n},\boldsymbol{\alpha},\theta) \propto \alpha_k^{x_{ki}}\, i^{1-x_{ki}} &\propto \left(\frac{\alpha_k}{\alpha_k+i}\right)^{x_{ki}}\left(\frac{i}{\alpha_k+i}\right)^{1-x_{ki}} \\ &\sim \mathrm{Bernoulli}\left(\frac{\alpha_k}{\alpha_k+i}\right) \end{aligned} \tag{B.11}$$

となります. この分布から補助変数 θ, x_{ki} をサンプリングしてから, 式(B.10)を使って α_k をサンプリングします. 実際には, 式(B.1)の $p(\mathbf{n}|\boldsymbol{\alpha})$ は N 個の文書について存在して, 尤度は式(5.79)になっていますから, 同様に計算すると, T_i を文書 i の長さとして

$$\text{(B.12)}\qquad \begin{cases} p(\theta_i|n,\boldsymbol{\alpha},x) & \sim \mathrm{Be}\left(\sum_k \alpha_k,\, T_i\right) \\ p(x_{ikj}|n,\boldsymbol{\alpha},\theta) & \sim \mathrm{Bernoulli}\left(\dfrac{\alpha_k}{\alpha_k+j}\right) \\ p(\alpha_k|n,\theta,x) & \sim \mathrm{Ga}\left(a+\sum_{i=1}^{N}\sum_{j=0}^{n(i,k)-1} x_{ikj},\, b-\sum_{i=1}^{N}\log\theta_i\right) \end{cases}$$

(**ディリクレ分布の $\boldsymbol{\alpha}$ のサンプリングの公式**)

で, $\boldsymbol{\alpha}$ をガンマ事後分布からサンプリングすることができます. □

η についても, 式(5.79)から同様にして計算すれば, 補助変数 ϕ_k, y_{kvj} を導入することで

$$\text{(B.13)}\qquad \begin{cases} p(\phi_k|m,\eta,y) & \sim \mathrm{Be}\left(V\eta,\, \sum_{v=1}^{V} m(k,v)\right) \\ p(y_{kvj}|m,\eta,\phi) & \sim \mathrm{Bernoulli}\left(\dfrac{\eta}{\eta+j}\right) \\ p(\eta|m,\phi,y) & \sim \mathrm{Ga}\left(a+\sum_{k=1}^{K}\sum_{v=1}^{V}\sum_{j=0}^{m(k,v)-1} y_{kvj},\right. \\ & \qquad \left. b-\sum_{k=1}^{K}\log\phi_k\right) \end{cases}$$

(**ディリクレ分布の η のサンプリングの公式**)

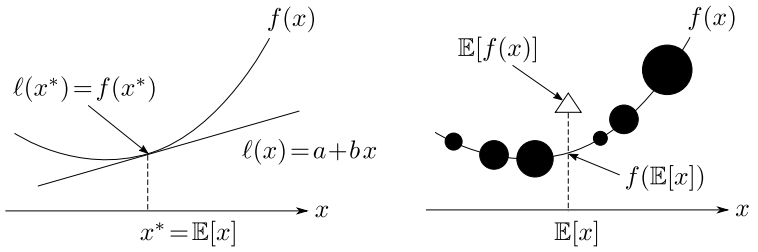
で, ガンマ事後分布からサンプリングを行えることがわかります. □

C　Jensen の不等式

Jensen(イェンセン)[*2] の不等式は, 統計学で広く使われる重要な不等式です. 以下, 図 45(a) のように実数 x の下に凸な関数 (凸関数) $f(x)$ があるとしましょう. 任意の確率分布 $p(x)$ を使ってとった x の期待値 $\mathbb{E}[x]=\sum_x p(x)\,x=x^*$ を接点にして $f(x)$ の接線 $\ell(x)=a+bx$ をとると, $f(x)$ は凸なので, つねに

$$\text{(C.1)}\qquad f(x)\geq a+bx$$

∗2　Johan Jensen (1859–1925) は, デンマーク工科大出身のエンジニア・数学者です.



(a) 凸関数と接線. 凸関数 $f(x)$ はつねに接線 $\ell(x)$ の上側にあり, $f(x)\geq\ell(x)$ が成り立っています.

(b) Jensen の不等式の物理的解釈. 重みづけられた球の重心△は, 必ず凸関数の上側にあります.

図 45: Jensen の不等式とその証明.

が成り立ちます. このとき, 接点 x^* では $\ell(x^*)=f(x^*)$ すなわち,

$$\ell(\mathbb{E}[x])=f(\mathbb{E}[x]) \tag{C.2}$$

になっていることに注意しましょう.

式(C.1)の両辺の期待値をとると, 期待値は不等式を保存するので [*3]

$$\begin{aligned}\mathbb{E}[f(x)] &\geq \mathbb{E}[a+bx] \\ &= a+b\,\mathbb{E}[x] \qquad \text{(期待値の線形性より)} \\ &= \ell(\mathbb{E}[x])=f(\mathbb{E}[x]) \qquad \text{(式(C.2)より)}\end{aligned} \tag{C.3}$$

となり, 一般に凸関数 $f(x)$ について

$$\mathbb{E}[f(x)]\geq f(\mathbb{E}[x]) \qquad (\textbf{Jensen の不等式}) \tag{C.4}$$

が成り立ちます. □

これは, 物理的には図 45(b) のように, 下に凸な曲線 $f(x)$ 上の各点に重み $p(x)$ に比例した大きさの球が乗っているとき, 球全体の重心△は必ず $f(x)$ の上側にある, ということを意味しています[30, p.35].

同様にして, 上に凸な関数 $f(x)$ について

---

*3 $f(x)\geq g(x)$ のとき, $\mathbb{E}[f(x)]=\sum_x p(x)f(x)\geq\sum_x p(x)g(x)=\mathbb{E}[g(x)]$ が成り立ちます.

$$f(\mathbb{E}[x]) \geq \mathbb{E}[f(x)]$$

が成り立ちますが, $y = \log x$ は図 2.18 に示したように上に凸なので, $x \to g(x)$ とおいて

$$\log \mathbb{E}[g(x)] \geq \mathbb{E}[\log g(x)]$$

すなわち,

$$\log \sum_x p(x)g(x) \geq \sum_x p(x) \log g(x) \tag{C.5}$$

(**Jensen の不等式 (対数の場合)**)

が成り立ちます. 式(C.5)を本書では, 5.2.2 節の EM アルゴリズムの証明に用いました.

アルゴリズムの一覧

公式の一覧

公式の一覧

公式の一覧

索引

さ

参考文献

[1] Chihiro Shibata, Kei Uchiumi, and Daichi Mochihashi. How LSTM Encodes Syntax: Exploring Context Vectors and Semi-Quantization on Natural Text. In *COLING 2020*, pages 4033–4043, 2020.

[2] Javier Ferrando, Gabriele Sarti, Arianna Bisazza, and Marta R. Costa-jussá. A Primer on the Inner Workings of Transformer-based Language Models. In *arXiv cs.CL*, 2024. https://arxiv.org/abs/2405.00208.

[3] André Martinet. *Éléments de linguistique générale, 5th Edition*. Armand Colin: Paris, 2008.

[4] 田中久美子. **記号と再帰 新装版: 記号論の形式・プログラムの必然**. 東京大学出版会, 2017.

[5] Sadao Kurohashi and Makoto Nagao. Building a Japanese Parsed Corpus while Improving the Parsing System. In *Proceedings of LREC 1998*, pages 719–724, 1998. https://nlp.ist.i.kyoto-u.ac.jp/?京都大学テキストコーパス.

[6] Jun Suzuki, Akinori Fujino, and Hideki Isozaki. Semi-Supervised Structured Output Learning Based on a Hybrid Generative and Discriminative Approach. In *Proceedings of EMNLP-CoNLL 2007*, pages 791–800, 2007.

[7] Jun Suzuki and Hideki Isozaki. Semi-Supervised Sequential Labeling and Segmentation Using Giga-Word Scale Unlabeled Data. In *ACL:HLT 2008*, pages 665–673, 2008.

[8] イリヤ・プリゴジン, イザベル・スタンジェール. **混沌からの秩序**. みすず書房, 1987. 伏見康治, 伏見譲, 松枝秀明 (訳).

[9] Hinrich Schütze. Dimensions of Meaning. In *Proceedings of Supercomputing'92*, pages 787–796, 1992.

[10] Christos H. Papadimitriou, Prabhakar Raghavan, Hisao Tamaki, and Santosh Vempala. Latent Semantic Indexing: A Probabilistic Analysis. *Journal of Computer and System Sciences*, 61(2):217–235, 2000.

[11] 石井公成. 仏教学における N-gram の活用. In **東京大学東洋文化研究所附属東洋学情報センター報「明日の東洋学」**, number 8, pages 2–4, 2002.

[12] 持橋大地, 大羽成征. **ガウス過程と機械学習**. 機械学習プロフェッショナルシリーズ. 講談社, 2019.

[13] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE.

Journal of Machine Learning Research, 9:2579–2605, 2008.
[14] 石野亜耶, 小早川健, 坂地泰紀, 嶋田和孝, 吉田光男. *Python* **ではじめるテキストアナリティクス入門**. 実践 Data Science シリーズ. 講談社, 2022.
[15] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
[16] Christopher D. Manning and Hinrich Schütze. **統計的自然言語処理の基礎**. 共立出版, 2017. 加藤恒昭, 菊井玄一郎, 林良彦, 森辰則 (訳).
[17] 北研二. **確率的言語モデル**. 言語と計算 4. 東京大学出版会, 1999.
[18] Daniel Jurafsky and James H. Martin. *Speech and Language Processing, 2nd Edition*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2008.
[19] 石井健一郎, 上田修功. **続・わかりやすいパターン認識—教師なし学習入門—**. オーム社, 2014.
[20] Zoubin Ghahramani. Unsupervised Learning. In *Advanced Lectures on Machine Learning. ML 2003. Lecture Notes in Computer Science, Vol.3176*, pages 72–112. Springer, 2004. https://doi.org/10.1007/978-3-540-28650-9_5.
[21] 高村大也. **言語処理のための機械学習入門**. 自然言語処理シリーズ. コロナ社, 2010.
[22] 佐藤坦. **はじめての確率論 測度から確率へ**. 共立出版, 1994.
[23] 清水泰隆. **統計学への確率論, その先へ：ゼロからの測度論的理解と漸近理論への架け橋 第** *2* **版**. 内田老鶴圃, 2021.
[24] Pierre-Simon Laplace. *Essai Philosophique sur les Probabilités*. 1814. 『確率の哲学的試論』内井惣七 (訳), 岩波文庫, 1997.
[25] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 623–656, 1948.
[26] Alastair J. Walker. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Transactions on Mathematical Software*, 31(3):253–256, 1977.
[27] Feras Saad, Cameron Freer, Martin Rinard, and Vikash Mansinghka. The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions. In *AISTATS 2020*, pages 1036–1046, 2020.
[28] David J Ward, Alan F Blackwell, and David J. C. MacKay. Dasher - a Data Entry Interface Using Continuous Gestures and Language Models. In *UIST 2000*, pages 129–137, 2000.
[29] Li Du, Lucas Torroba Hennigen, Tiago Pimentel, Clara Meister, Jason Eisner, and Ryan Cotterell. A Measure-Theoretic Characterization of Tight Language Models. In *ACL 2023*, pages 9744–9770, 2023.
[30] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.
[31] Daichi Mochihashi and Eiichiro Sumita. The Infinite Markov Model. In *Advances in Neural Information Processing Systems 20 (NIPS 2007)*, pages 1017–1024, 2008.
[32] 持橋大地, 隅田英一郎. 階層 Pitman-Yor 過程に基づく可変長 n-gram 言語モデル.

情報処理学会論文誌, 48(12):4023–4032, 2007.

[33] Daichi Mochihashi, Takeshi Yamada, and Naonori Ueda. Bayesian Unsupervised Word Segmentation with Nested Pitman-Yor Language Modeling. In *Proceedings of ACL-IJCNLP 2009*, pages 100–108, 2009.

[34] Andrei A. Markov. An Example of Statistical Investigation of the Text Eugene Onegin Concerning the Connection of Samples in Chains. In *Proceedings of the Academy of Sciences, St. Petersburg*, volume 7, pages 153–162, 1913.

[35] ロマーン・ヤーコブソン. **一般言語学**, **新装版**. みすず書房, 2019. 川本 他 (訳).

[36] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[37] 金子弘昌. *Python* **で学ぶ実験計画法入門 ベイズ最適化によるデータ解析**. KS 情報科学専門書. 講談社, 2021.

[38] 韓太舜, 小林欣吾. **情報と符号化の数理**. 培風館, 1999.

[39] Jihyeon Roh, Sang-Hoon Oh, and Soo-Young Lee. Unigram-Normalized Perplexity as a Language Model Performance Measure with Different Vocabulary Sizes. In *arXiv preprint*, 2020. https://arxiv.org/abs/2011.13220.

[40] S. Kullback and R. A. Leibler. On Information and Sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[41] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50:5–43, 2003.

[42] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory, 2nd Edition*. Wiley Series in Telecommunications. Wiley-Interscience, 2013.

[43] Tom Brown, Benjamin Mann, Nick Ryder, et al. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[44] Larry Wall, Tom Christiansen, and Jon Orwant. **プログラミング** *Perl* **第** *3* **版** *(Volume 1・2)*. オライリー・ジャパン, 2002. 近藤嘉雪 (訳).

[45] 山口和紀, 古瀬一隆. **新** *The UNIX Super Text (***上・下***)* **改訂増補版**. 技術評論社, 2003.

[46] Dale Dougherty and Arnold Robbins. *sed & awk* **プログラミング 改訂版**. A nutshell handbook. オライリー・ジャパン, 1997. 福崎俊博 (訳).

[47] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. **プログラミング言語** *AWK* **第** *2* **版**. オライリー・ジャパン, 2024. 千住治郎 (訳).

[48] Andrew Marc Greene. BASIX: An interpreter written in TEX. *TUGboat*, 11(3):381–392, 1990. https://www.ctan.org/tex-archive/macros/generic/basix.

[49] Kevin P. Murphy. *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

[50] C.M. Bishop, 元田, 栗田, 樋口, 松本, and 村田 (監訳). **パターン認識と機械学習**:**ベイズ理論による統計的予測**」*(***上・下***) (Pattern Recognition and Machine Learning)*. 丸善出版, 2012.

[51] Thomas M. Cover and Joy A. Thomas. **情報理論 —基礎と広がり—**. 共立出版, 2012. 山本博資, 古賀弘樹, 有村光晴, 岩本貢 (訳).
[52] 伊庭幸人.「情報」に関する 13 章 – 私家版・情報学入門 –. **物性研究**, 78(2):172–193, 2002. https://www.ism.ac.jp/~iba/a19.pdf.
[53] 水谷静夫. **言語と数学**. 数学ライブラリー〈教養篇〉5. 森北出版, 1970.
[54] 水谷静夫. **数理言語学**. 現代数学レクチャーズ D-3. 培風館, 1982.
[55] 近藤泰弘, 近藤みゆき. 平安時代古典語古典文学研究のための N-gram を用いた解析手法. In **言語情報処理学会第 7 回年次大会 発表論文集**, pages 209–212, 2001. https://anlp.jp/proceedings/annual_meeting/2001/pdf_dir/C3-4.pdf.
[56] 西村恕彦. **人文科学の** *FORTRAN 77*. 東京大学出版会, 1978.
[57] Folgert Karsdorp, Mike Kestemont, and Allen Riddell. *Humanities Data Analysis: Case Studies with Python*. Princeton University Press, 2021.
[58] 持橋大地, 山田武士, 上田修功. ベイズ階層言語モデルによる教師なし形態素解析. **情報処理学会研究報告** *2009-NL-190*, 2009.
[59] Yee Whye Teh. A Bayesian Interpretation of Interpolated Kneser-Ney. Technical Report TRA2/06, School of Computing, National University of Singapore, 2006.
[60] Linguistic Data Consortium. Web 1T 5-gram Version 1, 2006. https://catalog.ldc.upenn.edu/LDC2006T13.
[61] 言語資源協会. Web 日本語 N グラム第 1 版, 2007. https://www.gsk.or.jp/catalog/gsk2007-c.
[62] Henry Kučera and W. Nelson Francis. *Computational Analysis of Present-Day American English*. Brown University Press, 1967.
[63] Dirk Goldhahn, Thomas Eckart, and Uwe Quasthoff. Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages. In *LREC 2012*, pages 759–765, 2012.
[64] Harold Stanley Heaps. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, 1978.
[65] Gustav Herdan. *Type-Token Mathematics: A Textbook of Mathematical Linguistics*. Mouton en Company, 1960.
[66] George Kingsley Zipf. *The Psycho-Biology of Language: An Introduction to Dynamic Philology*. The MIT Press Classics Series. The MIT Press, 1935.
[67] Micheline Petruszewycz. L'histoire de la loi d'Estoup-Zipf : documents. *Mathématiques et Sciences humaines*, 44:41–56, 1973.
[68] Alain Lelu. Jean-Baptiste Estoup and the origins of Zipf's law: a stenographer with a scientific mind (1868-1950). *Boletín de Estadística e Investigación Operativa*, 30(1):66–77, 2014.
[69] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.
[70] 田中久美子. **言語とフラクタル：使用の集積の中にある偶然と必然**. 東京大学出版会, 2021.

[71] S. Naranan and V. K. Balasubrahmanyan. Models for Power Law Relations in Linguistics and Information Science. *Journal of Quantitative Linguistics*, 5(1-2):35–61, 1998.
[72] Martin Gerlach and Eduardo G. Altmann. Stochastic model for the vocabulary growth in natural languages. *Physical Review X*, 3:021006, 2013.
[73] Fadhel Ayed, Juho Lee, and Francois Caron. Beyond the Chinese Restaurant and Pitman-Yor processes: Statistical Models with double power-law behavior. In *ICML 2019*, pages 395–404, 2019.
[74] 林四郎. 臨時一語の構造. In **『国語学』第** *131* **集**, pages 15–26. 日本語学会, 1982.
[75] 佐藤敏紀, 橋本泰一, 奥村学. 単語分かち書き用辞書生成システム NEologd の運用 —文書分類を例にして —. **情報処理学会 自然言語処理研究会研究報告** *NL-229*, 2016.
[76] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, pages 3111–3119, 2013.
[77] Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jenifer C. Lai, and Robert L.Mercer. Class-Based n-gram Models of Natural Language. *Computational Linguistics*, 18(4):467–479, 1992.
[78] Kenneth Ward Church and Patrick Hanks. Word Association Norms, Mutual Information, and Lexicography. In *ACL 1989*, pages 76–83, 1989.
[79] Gerlof Bouma. Normalized (Pointwise) Mutual Information in Collocation Extraction. In *Proceedings of GSCL 30*, pages 31–40, 2009.
[80] Rameshwar D. Gupta and Donald St. P. Richards. The History of the Dirichlet and Liouville Distributions. *International Statistical Review*, 69(3):433–446, 2001.
[81] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986. http://www.nrbook.com/devroye/.
[82] David J. C. MacKay and Linda C. Bauman Peto. A Hierarchical Dirichlet Language Model. *Natural Language Engineering*, 1(3):289–308, 1995.
[83] Thomas P. Minka. Estimating a Dirichlet distribution, 2000. https://tminka.github.io/papers/dirichlet/.
[84] 佐藤一誠. **トピックモデルによる統計的潜在意味解析**. 自然言語処理シリーズ 8. コロナ社, 2015.
[85] Yee Whye Teh, Michael I. Jordan, Matthew J. Beal, and David M. Blei. Hierarchical Dirichlet Processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.
[86] Takeshi Kawabata and Masafumi Tamoto. Back-off Method for N-gram Smoothing based on Binomial Posteriori Distribution. In *ICASSP-96*, volume I, pages 192–195, 1996.
[87] Phil Cowans. *Probabilistic Document Modelling*. PhD thesis, University of Cambridge, 2006. http://www.inference.phy.cam.ac.uk/pjc51/thesis/index.html.
[88] Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language

modeling. In *Proceedings of ICASSP*, volume 1, pages 181–184, 1995.
[89] Stanley F. Chen and Joshua Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. Technical Report TR-10-98, Center for Research in Computing Technology, Harvard University, 1998.
[90] Graham Neubig and Chris Dyer. Generalizing and Hybridizing Count-based and Neural Language Models. In *EMNLP 2016*, pages 1163–1172, 2016.
[91] John Bridle. Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters. In *NIPS 1989*, pages 211–217, 1989.
[92] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems*, volume 13, 2000.
[93] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.
[94] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. *Distributed Representations*, pages 77–109. MIT Press, Cambridge, MA., 1986.
[95] J. S. Cramer. The Origins of Logistic Regression. Technical Report 2002-119/4, Tinbergen Institute, 2002.
[96] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In *NAACL 2013*, pages 746–751, 2013.
[97] Jun Suzuki and Masaaki Nagata. Right-truncatable Neural Word Embeddings. In *NAACL 2016*, pages 1145–1151, 2016.
[98] Omer Levy and Yoav Goldberg. Dependency-Based Word Embeddings. In *ACL 2014*, pages 302–308, 2014.
[99] Michael U. Gutmann and Aapo Hyvärinen. Noise-Contrastive Estimation of Unnormalized Statistical Models, with Applications to Natural Image Statistics. *Journal of Machine Learning Research*, 13(11):307–361, 2012.
[100] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.
[101] Omer Levy and Yoav Goldberg. Neural Word Embedding as Implicit Matrix Factorization. In *Advances in Neural Information Processing Systems 27*, pages 2177–2185, 2014.
[102] John A. Bullinaria and Joseph P. Levy. Extracting Semantic Representations from Word Co-occurrence Statistics: A Computational Study. *Behavior Research Methods*, 39:510–526, 2007.
[103] John A. Bullinaria and Joseph P. Levy. Extracting Semantic Representations from Word Co-occurrence Statistics: Stop-lists, Stemming and SVD. *Behavior Research Methods*, 44:890–907, 2012.
[104] G. ストラング. **線形代数とその応用**. 産業図書, 1978. 山口昌哉 (監訳) 井上昭 (訳).

[105] D. R. Cox. Regression Models and Life-Tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, 34(2):187–220, 1972.
[106] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global Vectors for Word Representation. In *EMNLP 2014*, pages 1532–1543, 2014.
[107] 横井祥, 下平英寿. 単語埋め込みの確率的等方化. In **言語処理学会第** *27* **回年次大会**, pages A7–1, 2021. https://www.anlp.jp/proceedings/annual_meeting/2021/pdf_dir/A7-1.pdf.
[108] Sho Yokoi, Han Bao, Hiroto Kurita, and Hidetoshi Shimodaira. Zipfian Whitening. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.
[109] Adriaan M. J. Schakel and Benjamin J. Wilson. Measuring Word Significance using Distributed Representations of Words. In *arXiv preprint*, 2015. https://arxiv.org/abs/1508.02297.
[110] 大山百々勢, 横井祥, 下平英寿. 単語ベクトルの長さは意味の強さを表す. In **言語処理学会第** *28* **回年次大会**, pages A5–1, 2022.
[111] Momose Oyama, Sho Yokoi, and Hidetoshi Shimodaira. Norm of Word Embedding Encodes Information Gain. In *EMNLP 2023*, page 2108–2130, 2023.
[112] Mikhail Khodak, Nikunj Saunshi, Yingyu Liang, Tengyu Ma, Brandon Stewart, and Sanjeev Arora. A La Carte Embedding: Cheap but Effective Induction of Semantic Feature Vectors. In *ACL 2018*, pages 12–22, 2018.
[113] 田中久美子. **言語とフラクタル*:* 使用の集積の中にある偶然と必然**. 東京大学出版会, 2021.
[114] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
[115] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.
[116] Joshua T. Goodman. A Bit of Progress in Language Modeling, Extended Version. Technical Report MSR–TR–2001–72, Microsoft Research, 2001.
[117] 岡崎直観, 荒瀬由紀, 鈴木潤, 鶴岡慶雅, 宮尾祐介. *IT Text* **自然言語処理の基礎**. オーム社, 2022.
[118] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool, 2017.
[119] Yoav Goldberg. **自然言語処理のための深層学習**. 共立出版, 2019. 加藤恒昭, 林良彦, 鷲尾光樹, 中林明子 (訳).
[120] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
[121] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. **深層学習**. KADOKAWA, 2018. 岩澤有祐, 鈴木雅大, 中山浩太郎, 松尾豊 (監訳).
[122] 高松瑞代. **応用がみえる線形代数**. 岩波書店, 2020.
[123] Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Alge-*

bra: Vectors, Matrices, and Least Squares. Cambridge University Press, 2018.
[124] H. S. Terrace, L. A. Petitto, R. J. Sanders, and T. G. Bever. Can an Ape Create a Sentence? *Science*, 206(4421):891–902, 1979.
[125] Toshitaka N. Suzuki. Animal linguistics: Exploring referentiality and compositionality in bird calls. *Ecological Research*, 36(2):221–231, 2021.
[126] Nipun Sadvilkar and Mark Neumann. PySBD: Pragmatic Sentence Boundary Disambiguation. In *Proceedings of Second Workshop for NLP Open Source Software (NLP-OSS)*, pages 110–114, 2020.
[127] Rachel Wicks and Matt Post. A unified approach to sentence segmentation of punctuated text in many languages. In *ACL 2021*, pages 3995–4007, 2021.
[128] 内海慶, 持橋大地. ベイズ教師なし文境界認識, 2025. 発表予定.
[129] Ryan Kiros, Yukun Zhu, Russ R. Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-Thought Vectors. In *NIPS 2015*, 2015.
[130] Prakhar Gupta, Matteo Pagliardini, and Martin Jaggi. Better Word Embeddings by Disentangling Contextual n-Gram Information. In *NAACL-HLT 2019*, pages 933–939, 2019.
[131] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR 2017*, 2017.
[132] Kawin Ethayarajh. Unsupervised Random Walk Sentence Embeddings: A Strong but Simple Baseline. In *Proceedings of The Third Workshop on Representation Learning for NLP*, pages 91–100, 2018.
[133] Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. A Latent Variable Model Approach to PMI-based Word Embeddings. *Transactions of the Association for Computational Linguistics*, 4:385–399, 2016.
[134] 伊庭幸人. **ベイズ統計と統計物理**. 岩波講座 物理の世界. 岩波書店, 2003.
[135] Mizuho Imada. Distribution of sentence length and dependency distance in children's compositions: Characteristics of natural language and variations in language development. *F1000Research*, 12(379), 2023.
[136] T. C. Mendenhall. The Characteristic Curves of Composition. *Science*, 9(214):237–249, 1887.
[137] G. Udny Yule. *The Statistical Study of Literary Vocabulary*. Cambridge University Press, 1944.
[138] C. B. Williams. A Note on the Statistical Analysis of Sentence-Length as a Criterion of Literary Style. *Biometrika*, 31(3/4):356–361, 1940.
[139] 石井正彦. 文の長さの統計モデル. **現代日本語研究**, 9:109–123, 2017.
[140] H. S. Sichel. On a Distribution Representing Sentence-Length in Written Prose. *Journal of the Royal Statistical Society. Series A (General)*, 137(1):25–34, 1974.
[141] Gábor Borbély and András Kornai. Sentence Length. In *16th Meeting on the Mathematics of Language*, pages 114–125, 2019.
[142] Gillian Z. Stein, Walter Zucchini, and June M. Juritz. Parameter Estimation for

the Sichel Distribution and Its Multivariate Extension. *Journal of the American Statistical Association*, 82(399):938–944, 1987.

[143] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*, pages 55–60, 2014. http://www.aclweb.org/anthology/P/P14/P14-5010.

[144] Nikita Kitaev and Dan Klein. Constituency Parsing with a Self-Attentive Encoder. In *ACL 2018*, pages 2676–2686, 2018.

[145] Tony A. Plate. Holographic Reduced Representations. *IEEE Transactions on Neural Networks*, 6(3):623–641, 1995.

[146] Ryosuke Yamaki, Tadahiro Taniguchi, and Daichi Mochihashi. Holographic CCG Parsing. In *ACL 2023*, pages 262–276, 2023.

[147] Mai Omura and Masayuki Asahara. UD-Japanese BCCWJ: Universal Dependencies Annotation for the Balanced Corpus of Contemporary Written Japanese. In *Second Workshop on Universal Dependencies (UDW 2018)*, pages 117–125, 2018.

[148] 松田寛. GiNZA — Universal Dependencies による実用的日本語解析 —. **自然言語処理**, 27(3):695–701, 2020.

[149] Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Bayesian Inference for PCFGs via Markov Chain Monte Carlo. In *Proceedings of HLT/NAACL 2007*, pages 139–146, 2007.

[150] J. ホップクロフト, R. モトワニ, J. ウルマン. **オートマトン 言語理論 計算論** *I* **[第** *2* **版]**. サイエンス社, 2003. 野崎昭弘, 高橋正子, 町田元, 山崎秀記 (訳).

[151] Xiang Hu, Pengyu Ji, Qingyang Zhu, Wei Wu, and Kewei Tu. Generative Pretrained Structured Transformers: Unsupervised Syntactic Language Models at Scale. In *ACL 2024*, pages 2640–2657, 2024.

[152] Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered Neurons: Integrating Tree Structures into Recurrent Neural Networks. In *arXiv preprint arXiv:1810.09536*, 2018.

[153] 能地宏. 文に隠れた構文構造を発見する統計モデル (特集「統計的言語研究の現在」). **統計数理**, 64(2):145–160, 2016.

[154] 持橋大地, 能地宏. 無限木構造隠れ Markov モデルによる階層的品詞の教師なし学習. **情報処理学会研究報告** *2016-NL-226*, 12:1–11, 2016.

[155] Julian Kupiec. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech & Language*, 6(3):225–242, 1992.

[156] Bernard Merialdo. Tagging English Text with a Probabilistic Model. *Computational linguistics*, 20(2):155–171, 1994.

[157] Sharon Goldwater and Tom Griffiths. A Fully Bayesian Approach to Unsupervised Part-of-Speech Tagging. In *Proceedings of ACL 2007*, pages 744–751, 2007.

[158] Jurgen Van Gael, Andreas Vlachos, and Zoubin Ghahramani. The infinite HMM for unsupervised PoS tagging. In *EMNLP 2009*, pages 678–687, 2009.

[159] Phil Blunsom and Trevor Cohn. A Hierarchical Pitman-Yor Process HMM for Unsupervised Part of Speech Induction. In *ACL 2011*, pages 865–874, 2011.
[160] Ryo Fujii, Ryo Domoto, and Daichi Mochihashi. Nonparametric Bayesian Semi-supervised Word Segmentation. *Transactions of ACL*, 5:179–189, 2017.
[161] 竹内孔一, 松本裕治. 隠れマルコフモデルによる日本語形態素解析のパラメータ推定. **情報処理学会論文誌**, 38(3):500–509, 1997.
[162] Andrew J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.
[163] G. David Forney Jr. The Viterbi Algorithm: A Personal History. In *arXiv preprint*, 2005. arXiv:cs/0504020 [cs.IT].
[164] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
[165] Stuart Geman and Donald Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.
[166] Daichi Mochihashi. Unbounded Slice Sampling. Technical Report Research Memorandum No.1209, The Institute of Statistical Mathematics, 2020. arXiv:2010.01760.
[167] Julian Besag. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society, Series B (Methodological)*, 48(3):259–279, 1986.
[168] Steven L. Scott. Bayesian Methods for Hidden Markov Models. *Journal of the American Statistical Association*, 97(457):337–351, 2002.
[169] Christophe Andrieu. On random- and systematic-scan samplers. *Biometrika*, 103(3):719–726, 2016.
[170] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. **ベイズデータ解析** *(**第** 3 **版**)*. 森北出版, 2024. 菅澤翔之助, 小林弦矢, 川久保友超, 栗栖大輔, 玉江大将, 株式会社 Nospare (訳).
[171] Jun S. Liu. The Collapsed Gibbs Sampler in Bayesian Computations with Applications to a Gene Regulation Problem. *Journal of the American Statistical Association*, 89(427):958–966, 1994.
[172] George Casella and Christian P. Robert. Rao-Blackwellisation of Sampling Schemes. *Biometrika*, 83(1):81–94, 1996.
[173] M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. The Infinite Hidden Markov Model. In *NIPS 2001*, pages 577–584, 2001.
[174] Marc Okrand. *The Klingon Dictionary: The Official Guide to Klingon Words and Phrases*. Gallery Books, 1992.
[175] Hitomi Yanaka and Koji Mineshima. Compositional Evaluation on Japanese Textual Entailment and Similarity. *Transactions of the Association for Computational Linguistics*, 10:1266–1284, 2022.
[176] David J. C. MacKay. Ensemble Learning for Hidden Markov Models. Technical

report, Cavendish Laboratory, University of Cambridge, 1997.
[177] Matthew J. Beal. *Variational Algorithms for Approximate Bayesian Inference*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003. https://cse.buffalo.edu/faculty/mbeal/thesis/.
[178] Kevin Knight. Bayesian Inference with Tears, 2009. https://kevincrawfordknight.github.io/papers/bayes-with-tears.pdf.
[179] Reina Akama, Kento Watanabe, Sho Yokoi, Sosuke Kobayashi, and Kentaro Inui. Unsupervised Learning of Style-sensitive Word Vectors. In *ACL 2018*, pages 572–578, 2018.
[180] Phillip Keung, Yichao Lu, György Szarvas, and Noah A. Smith. The Multilingual Amazon Reviews Corpus. In *EMNLP 2020*, pages 4563–4568, 2020.
[181] 株式会社ロンウィット. livedoor ニュースコーパス, 2014. http://www.rondhuit.com/download.html#ldcc.
[182] Hugo Larochelle and Iain Murray. The Neural Autoregressive Distribution Estimator. In *AISTATS 2011*, pages 29–37, 2011.
[183] 宮内裕人, 鈴木陽也, 秋山和輝, 梶原智之, 二宮崇, 武村紀子, 中島悠太, 長原一. 主観と客観の感情極性分類のための日本語データセット. In **言語処理学会第 *28* 回年次大会**, pages 1495–1499, 2022.
[184] Haruya Suzuki, Yuto Miyauchi, Kazuki Akiyama, Tomoyuki Kajiwara, Takashi Ninomiya, Noriko Takemura, Yuta Nakashima, and Hajime Nagahara. A Japanese Dataset for Subjective and Objective Sentiment Polarity Classification in Micro Blog Domain. In *LREC 2022*, pages 7022–7028, 2022.
[185] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
[186] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–22, 1977.
[187] 篠崎寿夫, 松森徳衛, 吉田正廣. **現代工学のための変分学入門**. 現代工学社, 1991.
[188] Greg C. G. Wei and Martin A. Tanner. A Monte Carlo Implementation of the EM Algorithm and the Poor Man's Data Augmentation Algorithms. *Journal of the American Statistical Association*, 85(411):699–704, 1990.
[189] ウラム. **数学のスーパースターたち*:* ウラムの自伝的回想**. 東京図書, 1979. 志村利雄 (訳).
[190] Eduardo G. Altmann, Janet B. Pierrehumbert, and Adilson E. Motter. Beyond Word Frequency: Bursts, Lulls, and Scaling in the Temporal Distributions of Words. *PLoS One*, 4(11):e7678, 2009.
[191] Kenneth W. Church. Empirical Estimates of Adaptation: The chance of Two Noriegas is closer to $p/2$ than p^2. In *COLING 2000*, 2000.
[192] K. Sjölander, K. Karplus, M. Brown, R. Hughey, A. Krogh, I.S. Mian, and D. Haussler. Dirichlet Mixtures: A Method for Improved Detection of Weak but Significant Protein Sequence Homology. *Computing Applications in the Bio-*

sciences, 12(4):327–345, 1996.
[193] 山本 幹雄, 貞光 九月, 三品 拓也. 混合ディリクレ分布を用いた文脈のモデル化と言語モデルへの応用. **情報処理学会研究報告** *2003-SLP-48*, pages 29–34, 2003.
[194] Issei Sato and Hiroshi Nakagawa. Topic models with power-law using Pitman-Yor process. In *KDD 2010*, page 673–682, 2010.
[195] 貞光九月, 待鳥裕介, 山本幹雄. 混合ディリクレ分布パラメータの階層ベイズモデルを用いたスムージング法. **情報処理学会研究報告** *2004-SLP-53*, pages 1–6, 2004.
[196] 山本幹雄, 持橋大地. Topic に基づく統計的言語モデルの最前線 –PLSI から HDP まで–. In **言語処理学会年次大会** *2006* **チュートリアル**, pages 11–28, 2006.
[197] Charles Elkan. Clustering Documents with an Exponential-Family Approximation of the Dirichlet Compound Multinomial Distribution. In *ICML 2006*, pages 289–296, 2006.
[198] 水谷静夫. 国語学に見る計量と数理. In **『数学セミナー』** *1993* **年** *10* **月号 特集***:* **数理化と計量化**, pages 30–34. 日本評論社, 1993.
[199] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. In *Neural Information Processing Systems 14*, 2001.
[200] Yee Whye Teh, David Newman, and Max Welling. A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, 2006.
[201] Tom Griffiths. Gibbs sampling in the generative model of Latent Dirichlet Allocation. Technical Report 11, Stanford University, 2002.
[202] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *PNAS*, 101:5228–5235, 2004.
[203] Thomas Minka and John Lafferty. Expectation-Propagation for the Generative Aspect Model. In *UAI 2002*, page 352–359, 2002.
[204] J. K. Pritchard, M. Stephens, and P. Donnelly. Inference of Population Structure Using Multilocus Genotype Data. *Genetics*, 155(2):945–959, 2000.
[205] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the Sampling Complexity of Topic Models. In *KDD 2014*, pages 891–900, 2014.
[206] Jinhui Yuan et al. LightLDA: Big Topic Models on Modest Computer Clusters. In *WWW 2015*, pages 1351–1361, 2015.
[207] Hanna M. Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. Evaluation Methods for Topic Models. In *ICML 2009*, pages 1105–1112, 2009.
[208] Wray Buntine and Aleks Jakulin. Applying Discrete PCA in Data Analysis. In *UAI 2004*, pages 59–66, 2004.
[209] Hanna M. Wallach, David Mimno, and Andrew McCallum. Rethinking LDA: Why Priors Matter. In *NIPS 2009*, pages 1973–1981, 2009.
[210] Jonathan Chang, Sean Gerrish, Chong Wang, Jordan Boyd-graber, and David Blei. Reading Tea Leaves: How Humans Interpret Topic Models. In *NIPS 2009*, 2009.
[211] Jey Han Lau, David Newman, and Timothy Baldwin. Machine Reading Tea

Leaves: Automatically Evaluating Topic Coherence and Topic Model Quality. In *EACL 2014*, pages 530–539, 2014.

[212] Feng Nan, Ran Ding, Ramesh Nallapati, and Bing Xiang. Topic Modeling with Wasserstein Autoencoders. In *ACL 2019*, pages 6345–6381, 2019.

[213] Shusei Eshima, Kosuke Imai, and Tomoya Sasaki. Keyword-Assisted Topic Models. *American Journal of Political Science*, 68(2):730–750, 2024.

[214] Jacob Eisenstein, Amr Ahmed, and Eric P. Xing. Sparse Additive Generative Models of Text. In *ICML 2011*, pages 1041–1048, 2011.

[215] Margaret E. Roberts, Brandon M. Stewart, Dustin Tingley, Christopher Lucas, Jetson Leder-Luis, Shana Kushner Gadarian, Bethany Albertson, and David G. Rand. Structural Topic Models for Open-Ended Survey Responses. *American Journal of Political Science*, 58(4):1064–1082, 2014.

[216] Adji B. Dieng, Francisco J. R. Ruiz, and David M. Blei. Topic Modeling in Embedding Spaces. *Transactions of the Association for Computational Linguistics*, 8:439–453, 2020.

[217] Shusei Eshima and Daichi Mochihashi. Scale-invariant Infinite Hierarchical Topic Model. In *Findings of ACL 2023*, pages 11731–11746, 2023.

[218] David M. Blei and Jon D. Mcauliffe. Supervised Topic Models. In *Advances in Neural Information Processing Systems 20*, pages 121–128, 2008.

[219] Daniel Lee and H. Sebastian Seung. Algorithms for Non-negative Matrix Factorization. In *Advances in Neural Information Processing Systems*, volume 13, 2000.

[220] John Canny. GaP: A Factor Model for Discrete Data. In *SIGIR 2004*, pages 122–129, 2004.

[221] 持橋大地. GaP, NMF, and more, 2006. http://chasen.org/˜daiti-m/paper/gap-nmf.pdf.

[222] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[223] Thomas Hofmann. Probabilistic Latent Semantic Indexing. In *SIGIR 1999*, pages 50–57, 1999.

[224] Quoc Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. In *ICML 2014*, pages 1188–1196, 2014.

[225] Daichi Mochihashi. Researcher2Vec: Neural Linear Model of Scholar Recommendation for Funding Agency. In *International Society for Scientometrics and Informatics (ISSI 2023)*, 2023.

[226] G. Salton and C. S. Yang. On the specification of term values in automatic indexing. *Journal of Documentation*, 29:351–372, 1973.

[227] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[228] Kishore Papineni. Why Inverse Document Frequency? In *NAACL 2001*, pages

1–8, 2001.
[229] 南風原朝和. **心理統計学の基礎：統合的理解のために**. 有斐閣アルマ. 有斐閣, 2002.
[230] Sungjoon Park, JinYeong Bak, and Alice Oh. Rotated Word Vector Representations and their Interpretability. In *EMNLP 2017*, pages 401–411, 2017.
[231] Hiroaki Yamagiwa, Momose Oyama, and Hidetoshi Shimodaira. Discovering Universal Geometry in Embeddings with ICA. In *EMNLP 2023*, pages 4647–4675, 2023.
[232] 東京大学教養学部統計学教室 (編). **統計学入門**. 基礎統計学 I. 東京大学出版会, 1991.
[233] Sascha Rothe, Sebastian Ebert, and Hinrich Schütze. Ultradense Word Embeddings by Orthogonal Transformation. In *NAACL 2016*, pages 767–777, 2016.
[234] Bianca Zadrozny and Charles Elkan. Transforming Classifier Scores into Accurate Multiclass Probability Estimates. In *KDD 2002*, pages 694–699, 2002.
[235] Michael Laver, Kenneth Benoit, and John Garry. Extracting Policy Positions from Political Texts Using Words as Data. *American Political Science Review*, 97(2):311–331, 2003.
[236] Jonathan B. Slapin and Sven-Oliver Proksch. A Scaling Model for Estimating Time-Series Party Positions from Texts. *American Journal of Political Science*, 52(3):705–722, 2008.
[237] 持橋大地. 確率的潜在意味スケーリング. In **情報処理学会研究報告** *2021-NL-249*, number 9, pages 1–16, 2021.
[238] Kohei Watanabe. Latent Semantic Scaling: A Semisupervised Text Analysis Technique for New Domains and Languages. *Communication Methods and Measures*, 15:81–102, 2020.
[239] Philipp Dufter and Hinrich Schütze. Analytical Methods for Interpretable Ultradense Word Embeddings. In *EMNLP-IJCNLP 2019*, pages 1185–1191, 2019.
[240] Ikuya Yamada, Akari Asai, Jin Sakuma, Hiroyuki Shindo, Hideaki Takeda, Yoshiyasu Takefuji, and Yuji Matsumoto. Wikipedia2Vec: An Efficient Toolkit for Learning and Visualizing the Embeddings of Words and Entities from Wikipedia. In *EMNLP 2020: System Demonstrations*, pages 23–30, 2020.
[241] Malay Ghosh. Inconsistent maximum likelihood estimators for the Rasch model. *Statistics & Probability Letters*, 23(2):165–170, 1995.
[242] Radford M. Neal. *MCMC Using Hamiltonian Dynamics*. Chapman and Hall/CRC, 2011.
[243] Qing Liu and Donald A. Pierce. A Note on Gauss-Hermite Quadrature. *Biometrika*, 81(3):624–629, 1994.
[244] ピーター・D・ホフ. **標準ベイズ統計学**. 朝倉書店, 2022. 入江薫, 菅澤翔之助, 橋本真太郎 (訳).
[245] Peter D. Hoff. *A First Course in Bayesian Statistical Methods*. Springer Texts in Statistics. Springer, 2009.
[246] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC, 2013. http://

www.stat.columbia.edu/˜gelman/book/.
[247] George Casella and Roger L. Berger. *Statistical Inference, Second Edition*. Texts in Statistical Science. Chapman & Hall/CRC, 2024.
[248] Peter M. Lee. *Bayesian Statistics: An Introduction, 4th Edition*. Wiley, 2012.
[249] Radford M. Neal and Geoffrey E. Hinton. *A View of the EM Algorithm that Justifies Incremental, Sparse, and other Variants*, pages 355–368. Dordrecht: Kluwer Academic Publishers, 1998.
[250] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. In *arXiv preprint*, 2013. arXiv:1312.6114.
[251] 岩田具治. **トピックモデル**. 機械学習プロフェッショナルシリーズ. 講談社, 2015.
[252] Kosuke Imai. *Quantitative Social Science: An Introduction*. Princeton University Press, 2017.
[253] 今井耕介. **社会科学のためのデータ分析入門** *(***上・下***)*. 岩波書店, 2018. 粕谷祐子, 原田勝孝, 久保浩樹 (訳).
[254] Justin Grimmer, Margaret E. Roberts, and Brandon M. Stewart. *Text as Data: A New Framework for Machine Learning and the Social Sciences*. Princeton University Press, 2022.
[255] Matthew Gentzkow, Bryan Kelly, and Matt Taddy. Text as Data. *Journal of Economic Literature*, 57(3):535–574, 2019.
[256] Elliott Ash and Stephen Hansen. Text Algorithms in Economics. *Annual Review of Economics*, 15:659–688, 2023.
[257] Viet-An Nguyen, Jordan Boyd-Graber, Philip Resnik, and Kristina Miler. Tea Party in the House: A Hierarchical Ideal Point Topic Model and Its Application to Republican Legislators in the 112th Congress. In *ACL 2015*, pages 1438–1448, 2015.
[258] 持橋大地. 生きた言葉をモデル化する—自然言語処理と数学の接点. **『数学セミナー』** *2007* **年** *11* **月号**, pages 37–43, 2007.