

Scalaで実装するパターン認識と機械学習



2023年3月18日

無線部開発班 JG1VPP

目次

第1章 初歩的な機械学習モデル	3
1.1 線型回帰	4
1.2 単純ベイズ分類器	5
第2章 ニューラルネットワーク	6
2.1 誤差逆伝播法の理論	7
2.2 誤差逆伝播法の実装	8
2.3 ソフトマックス関数	9
2.4 鞍点と学習率の調整	10
2.5 通時的誤差逆伝播法	11
第3章 サポートベクターマシン	12
3.1 双対問題の導出	13
3.2 逐次的な最適化	13
3.3 線型分離の学習	14
3.4 特徴空間の変換	15
3.5 ヒルベルト空間	16
第4章 決定木の学習と汎化性能	17
4.1 情報源符号化定理	17
4.2 条件分岐の最適化	18
4.3 アンサンブル学習	19
4.4 ブースティング法	19
4.5 汎化性能の最適化	20
4.6 圧縮アルゴリズム	21
第5章 潜在的ディリクレ配分法	22
5.1 確率的潜在意味解析	22
5.2 潜在的な話題の学習	23
5.3 単語の類似度の推定	23
第6章 混合正規分布と最尤推定	24
6.1 クラスタリングの実装	25
6.2 期待値最大化法の理論	26
6.3 期待値最大化法の実装	27
6.4 変分ベイズ推定の理論	28
6.5 母数の事前分布の設定	29
6.6 母数の事後分布の導出	30
6.7 変分ベイズ推定の実装	31

第1章 初歩的な機械学習モデル

機械学習とは、説明変数 x と目的変数 y の組 (x, y) の集合 \mathbb{T} から、変数 x, y の関係を表す関数 f を推定する方法である。集合 \mathbb{T} が、関数 f の取る値 y を具体的に列挙する場合は、その問題を教師あり学習と呼び、集合 \mathbb{T} を教師データと呼ぶ。

$$\forall x, y: (x, y) \in \mathbb{T} \Rightarrow y \approx f(x). \quad (1.1)$$

教師あり学習で、目的変数 y が、**クラス** と呼ばれる離散値を取る場合を**分類**と呼ぶ。その初歩的な例が**最近傍法**である。最近傍法では、未知の点 x のクラスは、 x の至近距離にある既知の K 個の点の多数決で決まる。Fig. 1.1(a) に例を示す。

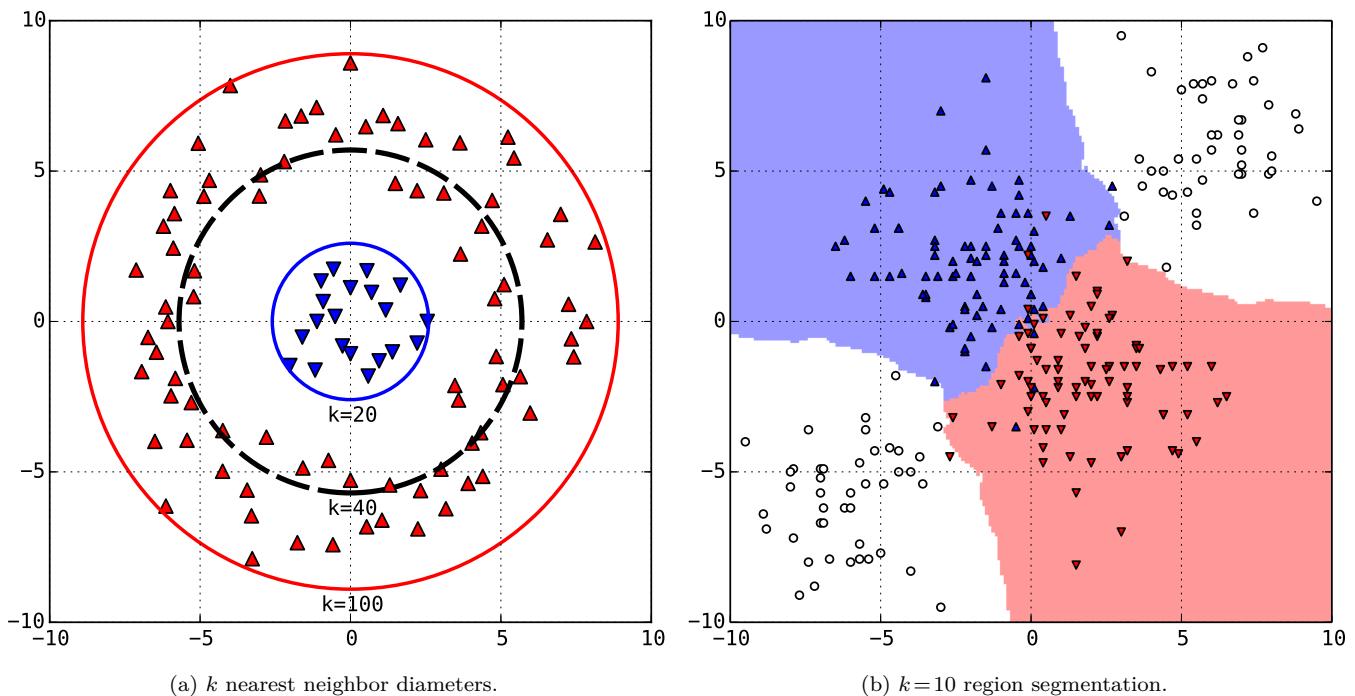


Fig. 1.1: k nearest neighbor model.

Fig. 1.1(b) は、各々が正規分布に従う 3 クラスの点の集合を学習し、空間全体をそれらのクラスに分類した結果である。最近傍法では、適当な距離関数 d を使う。距離関数 d は、式 (1.2) の**距離の公理**を満たし、任意の 2 点の距離を定義する。

$$\forall x, y, z \in \mathbb{R}^D: 0 \leq d(x, y) = d(y, x) \leq d(x, z) + d(z, y), x = y \Leftrightarrow d(x, y) = 0. \quad (1.2)$$

最近傍法は、他の著名な教師あり学習の手法と比較して、事前の学習が不要である点が特徴的で、**遅延学習** と呼ばれる。以上の議論に基づき、最近傍法を実装しよう。引数は、参照する近傍点の個数 K と、集合 $\{(x, y)\}$ と、距離関数 d である。

```
class KNN[D, T](k: Int, data: Seq[(D, T)], d: (D, D) => Double) {
  def apply(x: D) = data.sortBy((p, t) => d(x, p)).take(k).groupBy((p, t) => t).maxBy((g, s) => s.size)._1
}
```

使用例を以下に示す。距離関数の例として、初等幾何学の基礎であるユークリッド距離や**マンハッタン距離**を使用した。後者は、座標の差の絶対値の総和を距離とする。距離関数の最適な選択肢は、分類対象の問題の性質に応じて変化する。

```
val samples2d = Seq.fill(100)(Seq.fill(2)(util.Random.nextGaussian) -> util.Random.nextBoolean)
val euclidean = new KNN(5, samples2d, (a, b) => math.sqrt(a.zip(b).map((a, b) => (a-b)*(a-b)).sum))
val manhattan = new KNN(5, samples2d, (a, b) => a.zip(b).map(_-_).map(_.abs).sum)
```

1.1 線型回帰

教師あり学習で、目的変数 y が連続な値を取る場合を回帰と呼ぶ。その代表的な例が、**線型回帰**である。式 (1.3) に示す。適当な基底関数 ϕ の線型結合である。基底関数は、関数 f の形に応じて選ぶ。例えば、多項式基底やガウス基底を使う。

$$\mathbf{y} + \varepsilon \approx f(\mathbf{x}) = \sum_{k=0}^K w_k \phi_k(\mathbf{x}) = {}^t \mathbf{w} \phi(\mathbf{x}). \quad (1.3)$$

基本的に、変数 x, y は誤差 ε を含む。例えば、映像や音声信号には、式 (1.4) に示す、分散 σ^2 の**ガウスノイズ**が重畠する。

$$y \sim \mathcal{L}(f) = p(y | \mathbf{x}, f) = \mathcal{N}(y | f(\mathbf{x}), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left\{ -\frac{(y - f(\mathbf{x}))^2}{2\sigma^2} \right\}. \quad (1.4)$$

式 (1.4) の確率は、確率 f の妥当性と見做せる。これを**尤度**と呼ぶ。尤度の最大値を探せば、最適な関数 \hat{f} が推定できる。これを**最尤推定**と呼び、機械学習の基本原理である。尤度の対数から、式 (1.5) が導出される。関数 E を 2 乗誤差と呼ぶ。

$$\hat{f} = \arg \max_f \log p(y | \mathbf{x}, f) = \arg \min_f \{y - f(\mathbf{x})\}^2 = \arg \min_f E(\mathbf{w}). \quad (1.5)$$

誤差 E を削減する方向に加重 \mathbf{w} を動かす操作を繰り返すと、極小点に収束する。これを**勾配法**と呼ぶ。式 (1.6) に示す。

$$\hat{\mathbf{w}} = \mathbf{w} - \eta \nabla E(\mathbf{w}) = \mathbf{w} + \eta \sum_{n=1}^N \{y_n - {}^t \mathbf{w} \phi(\mathbf{x}_n)\} \phi(\mathbf{x}_n), \text{ where } \eta \ll \left| \frac{\mathbf{w}}{\nabla E(\mathbf{w})} \right|. \quad (1.6)$$

定数 η を**学習率**と呼ぶ。以上の議論に基づき、線型回帰を実装する。引数は、学習率 η と、集合 $\{x, y\}$ と、基底 Φ である。

```
class Regression(e: Double, data: Seq[(Double, Double)], p: Seq[Double=>Double], epochs: Int = 1000) {
  val w = Array.fill[Double](p.size)(0)
  def apply(x: Double) = w.zip(p.map(_(x))).map(_ * _).sum
  for(n <- 1 to epochs; (x,y) <- data) w.zip(p).map(_ + e * (y - this(x)) * _(x)).copyToArray(w)
}
```

Fig. 1.2 は、多項式基底とガウス基底を利用して、各々の基底に適した形状の曲線に対し、線型回帰を行った結果である。

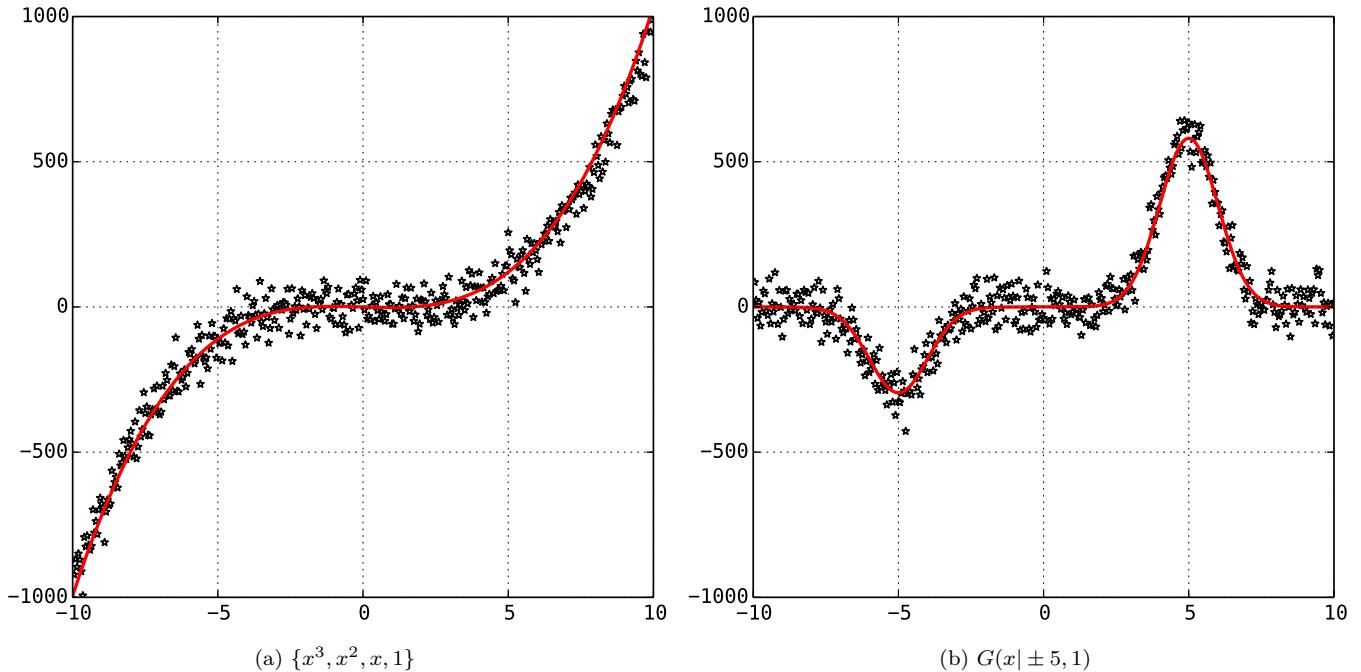


Fig. 1.2: linear basis function model.

なお、加重 \mathbf{w} は最適化されたが、基底関数 Φ 自体は最適化されず、**ハイパーパラメータ**として扱った点に注意を要する。

1.2 単純ベイズ分類器

自然言語で記述された記事の話題を分類する問題を考える。記事 d は単語 w_n の列であり、単語は話題 c から生成される。記事 d の内容を話題 c と仮定する。この仮説の尤度は、単語の共起を無視すれば、式 (1.7) の条件付き確率で定義される。

$$\mathcal{L}(c) = P(d|c) = P(w_1, \dots, w_{N_d} | c) = \prod_{n=1}^{N_d} P(w_n | c, w_1, \dots, w_{n-1}) \simeq \prod_{n=1}^{N_d} P(w_n | c). \quad (1.7)$$

最適な話題 \hat{c} は、式 (1.8) の条件付き確率を最大化する。確率 $P(c)$ は記事 d とは独立した確率で、**事前確率**と呼ばれる。式 (1.8) は、記事 d を観測した後の話題 c の確率で、これを**事後確率**と呼ぶ。式 (1.8) の変形は、**ベイズの定理**を使った。

$$\hat{c} = \arg \max_c P(c|d) = \arg \max_c \frac{P(c) P(d|c)}{P(d)} = \arg \max_c P(c) P(d|c) = \arg \max_c P(c) \prod_{n=1}^{N_d} P(w_n | c). \quad (1.8)$$

ただし、初めて出現した単語 w に対して、式 (1.8) の確率が 0 になる事態を防ぐため、式 (1.9) の**ラプラス平滑化**を行う。

$$P(w|c) = \frac{P(w,c)}{P(c)} \simeq \frac{N_{wc} + 1}{N_c + 1} > 0, \Leftrightarrow P(w) = \frac{1}{|V|}, \text{ where } N_c = \sum_{w \in V} N_{wc}. \quad (1.9)$$

変数 N_{wc} は、組 (w, c) の頻度である。式 (1.9) は、単語 w の事前確率を第5章で学ぶディリクレ分布と仮定して導かれる。式 (1.8) の分類器を**単純ベイズ分類器**と呼ぶ。以下に実装を示す。引数は、既知の記事の列と、対応する話題の列である。

```
class NaiveBayes[D<:Seq[W], W, C](texts: Seq[D], classes: Seq[C]) {
  val nw = scala.collection.mutable.Map[(W,C),Double]().withDefaultValue(1)
  val pc = classes.groupBy(identity).map(_ -> _.size.toDouble / texts.size)
  def pwc(c: C)(w: W) = nw(w,c) / texts.flatten.distinct.map(nw(_,c)).sum
  def pcd(d: D)(c: C) = math.log(pc(c)) + d.map(pwc(c)).map(math.log).sum
  def apply(d: D) = classes.distinct.maxBy(pcd(d))
  for((d,c) <- texts.zip(classes); w <- d) nw(w,c) += 1
}
```

Fig. 1.3(a) は、百科事典で各地方の記事から固有名詞を抽出して学習し、都道府県の記事の地方を推定した結果である。

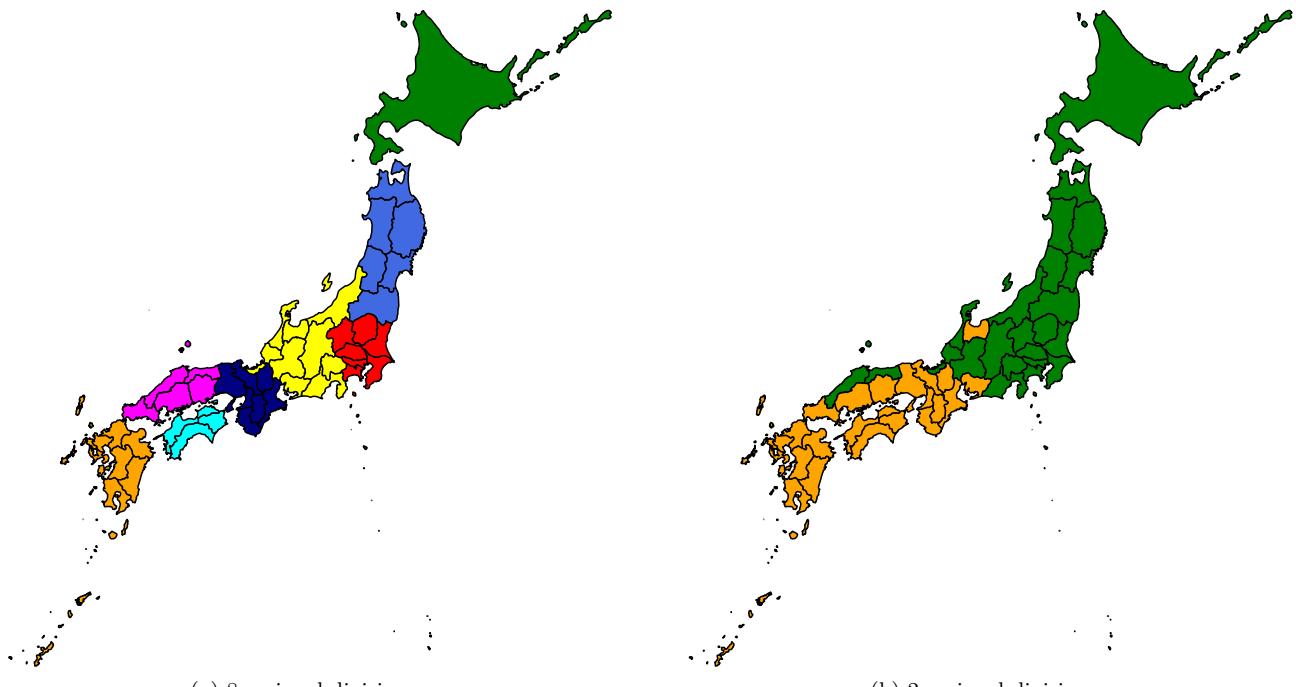


Fig. 1.3: Japanese map division into regions based on classification of Wikipedia pages.

Fig. 1.3(b) は、東日本と西日本の記事を学習して、都道府県を分類した結果である。単純だが、高精度な分類ができる。

第2章 ニューラルネットワーク

ニューラルネットワークは、線型回帰に似た**ニューロン**と呼ばれる関数を連結して、連鎖構造にした複雑な関数である。単体のニューロンは、線型回帰の後に、**活性化関数**と呼ばれる非線型な関数 f を適用した関数で、式 (2.1) で定義される。

$$\mathbf{y} \simeq f(\mathbf{z}) = f(W\mathbf{x}). \quad (2.1)$$

単体では、線型回帰と同じ程度の表現能力だが、何層も重ねることで、任意の**滑らかな関数**を任意の精度で近似できる。循環構造がなく、直線的な構造の**順伝播型**の動作は、式 (2.2) の漸化式で定義できる。循環構造の場合は第 2.5 節で扱う。

$$\mathbf{y}_n = \mathbf{x}_{n+1} = f_n(\mathbf{z}_n) = f_n(W_n \mathbf{x}_n). \quad (2.2)$$

式 (2.2) で、第 n 層は前の層から値 \mathbf{x}_n を受容し、行列 W_n で加重して活性化関数 f_n を適用し、後続の層に値 \mathbf{y}_n を渡す。活性化関数には、**シグモイド関数**が広く利用される。式 (2.3) に定義する。これは、2 クラスの分類器のように振る舞う。

$$f_{\text{sigm}}(z) = \frac{1}{1 + e^{-z}} = \frac{1}{2} \tanh \frac{z}{2} + \frac{1}{2}. \quad (2.3)$$

代表的な活性化関数の例を Fig. 2.1(a) に示す。他には、式 (2.4) に示す**ソフトマックス関数**も特に最終層で利用される。

$$y \sim \hat{p}(y) = f_{\text{softmax}}(\mathbf{z}) = \frac{1}{e^{z_1} + \dots + e^{z_K}} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{pmatrix}. \quad (2.4)$$

最終層の活性化関数を適切に選ぶと、回帰や分類など、様々な問題に対応できる。特に分類問題の例は第 2.3 節に述べる。Fig. 2.1(b) は、活性化関数にシグモイド関数を利用し、論理和を求める例である。直線 $f(\mathbf{x}) = 0.5$ は分類の境界を表す。

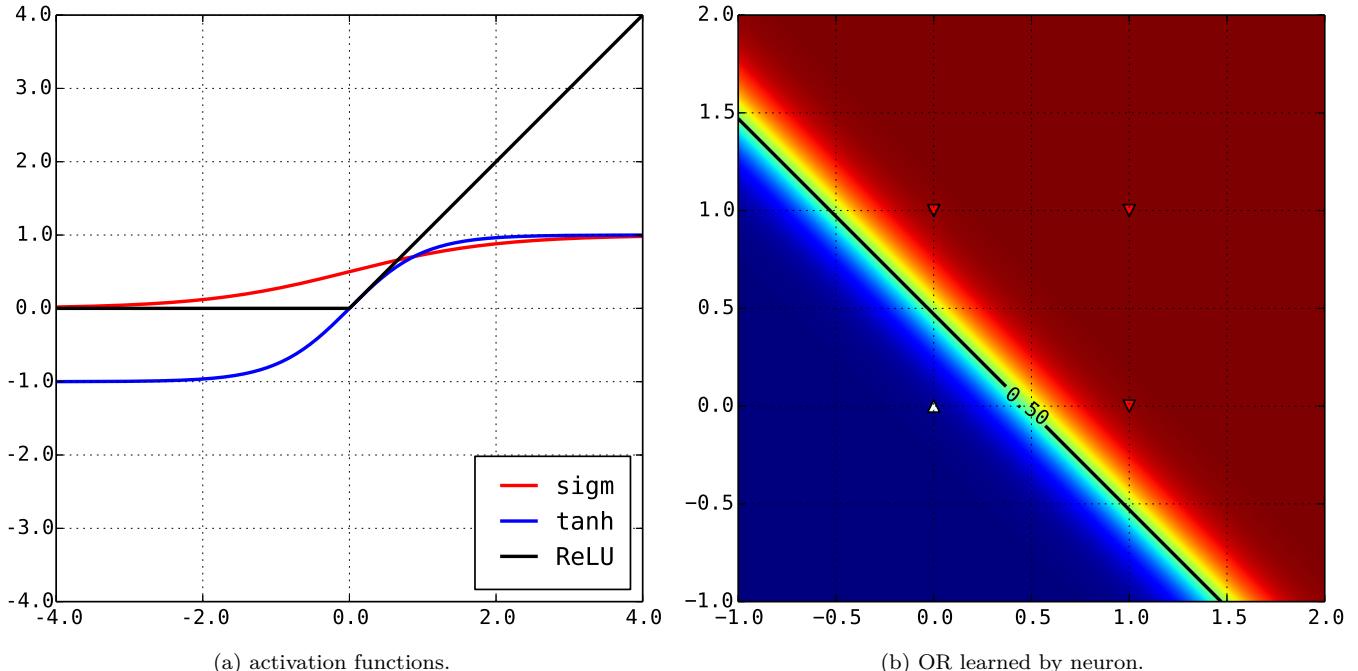


Fig. 2.1: neuron mechanism.

論理和や論理積は、分類の境界が直線や超平面となる単純な問題で、これを**線型分離可能**と呼び、単層でも表現できる。第 2 章で学ぶ誤差逆伝搬法は、多数の層を訓練して、線型分離が困難な問題に適合させる**深層学習**を支える技法である。

2.1 誤差逆伝播法の理論

深層学習では、多数の層の加重を最適化して、誤差 E を最小化する。最適解の計算は困難なので、逐次的に最適化する。具体的な手順は、以下の通りである。まず、第 n 層の加重 W_n を最適化の対象とし、式 (2.5) に示す勾配法で最適化する。

$$w_n^{ij} = w_n^{ij} - \eta \frac{\partial E}{\partial w_n^{ij}} = w_n^{ij} - \eta \frac{\partial z_n^j}{\partial w_n^{ij}} \frac{\partial x_{n+1}^j}{\partial z_n^j} \frac{\partial E}{\partial x_{n+1}^j} = w_n^{ij} - \eta x_n^i \frac{\partial f}{\partial z_n^j}(z_n^j) \frac{\partial E}{\partial x_{n+1}^j}. \quad (2.5)$$

定数 η は学習率で、変数 x_n^i, z_n^j は、変数 x_n, z_n の第 i, j 成分である。さて、式 (2.2) から式 (2.6) の漸化式が導出される。式 (2.6) の漸化式を利用して、誤差 E を逆方向に伝播させ、式 (2.5) の最適化を各層で行う。これを**誤差逆伝播法**と呼ぶ。

$$\frac{\partial E}{\partial x_n^i} = \sum_{j=1}^J \frac{\partial z_n^j}{\partial x_n^i} \frac{\partial x_{n+1}^j}{\partial z_n^j} \frac{\partial E}{\partial x_{n+1}^j} = \sum_{j=1}^J w_n^{ij} \frac{\partial f}{\partial z_n^j}(z_n^j) \frac{\partial E}{\partial x_{n+1}^j}. \quad (2.6)$$

漸化式の初期値を考える。2乗誤差関数 E_{sq} を仮定すると、最終層の値 \hat{y} と目的変数 y に対し、導関数は式 (2.7) になる。

$$\frac{\partial E_{\text{sq}}}{\partial \hat{y}^j} = \frac{\partial}{\partial \hat{y}^j} \frac{1}{2} \|y - \hat{y}\|^2, \text{ where } E_{\text{sq}}(\hat{y}, y) = \frac{1}{2} \|y - \hat{y}\|^2. \quad (2.7)$$

式 (2.6) には、活性化関数の微分が含まれるが、活性化関数は巧妙に設計されており、実に単純な四則演算で計算できる。

$$\frac{\partial f_{\text{sigm}}}{\partial z_n^j}(z_n^j) = \frac{e^{-z_n^j}}{(1 + e^{-z_n^j})^2} = x_{n+1}^j(1 - x_{n+1}^j). \quad (2.8)$$

第2.2節では、誤差逆伝播法を備えると同時に、自在に層構造を定義可能な深層学習を実装する。利用方法を以下に示す。

```
val model3 = new Output(1, _-_)
val model2 = new Offset(3, new Sigmoid, ()=>new PlainSGD, model3)
val model1 = new Offset(2, new Sigmoid, ()=>new PlainSGD, model2)
for(n <- 1 to 1000000; x <- 0 to 1; y <- 0 to 1) model1.bp(Seq(x,y), Seq(x^y))
```

複数の非線型変換を持つ恩恵で、線型分離が困難な分類問題にも対応できる。具体例として、排他的論理和を学習する。通常の結果を Fig. 2.2(a) に、各層の変数 x に定数項を含む場合の結果を (b) に示す。定数項の有無で、境界が変化した。

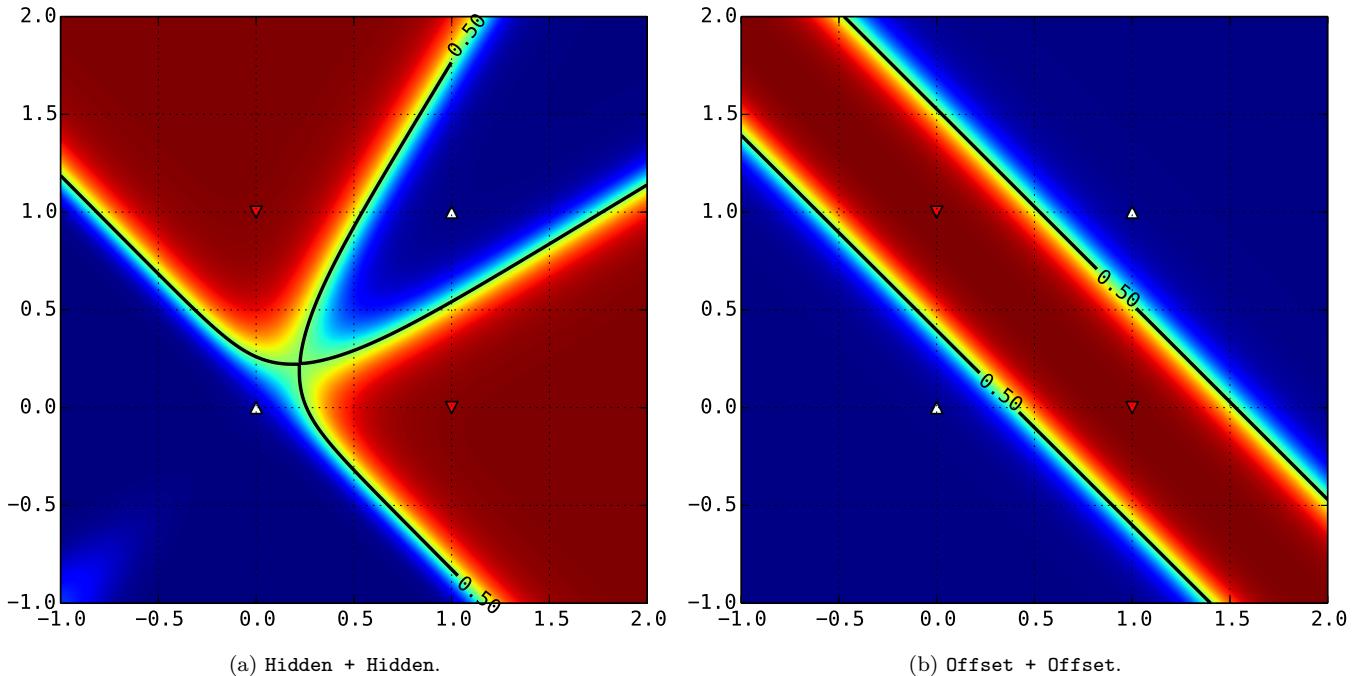


Fig. 2.2: exclusive OR learned by a three-layer perceptron.

ぜひ実装して、学習の途中経過を観察しよう。また、加重の初期値によって、収束までの時間が変わるものも観察できる。

2.2 誤差逆伝播法の実装

以上の数式を実装し、誤差 E を逆伝播させ、最終層から最初層まで逐次的に最適化しよう。最初に、勾配法を定義する。

```
abstract class SGD(var w: Double = math.random) extends (Double => Unit)
```

これは、加重 w の抽象化であり、式 (2.5) による最適化も行う。具体的な最適化の手順は、勾配法を継承して実装する。

```
class PlainSGD(e: Double = 0.01) extends SGD {
  def apply(dE: Double): Unit = this.w -= e * dE
}
```

引数は、学習率 η である。学習時は、誤差 E の勾配 ∇E を受け取り、加重 w を修正する。次に、活性化関数を定義する。

```
trait Act {
  def fp(z: Seq[Double]): Seq[Double]
  def bp(y: Seq[Double]): Seq[Double]
}
```

活性化関数は、順伝播と逆伝播を行う。以下に、具体的な実装例を示す。順伝播は式 (2.3) に、逆伝播は式 (2.8) に従う。

```
class Sigmoid extends Act {
  def fp(z: Seq[Double]) = z.map(z => 1 / (1 + math.exp(-z)))
  def bp(z: Seq[Double]) = this.fp(z).map(y => y * (1.0 - y))
}
```

次に、層を定義する。加重と活性化関数を持ち、順伝播と逆伝播を行う中間層と、誤差関数を計算する最終層が派生する。

```
abstract class Neuron(val dim: Int) {
  def fp(x: Seq[Double]): Seq[Double]
  def bp(x: Seq[Double], t: Seq[Double]): Seq[Double]
}
```

最終層を実装する。引数は、最終層が出力する値の要素数と、誤差の導関数である。誤差関数は、**損失関数**とも呼ばれる。

```
class Output(dim: Int = 1, loss: (Double, Double) => Double = _ - _) extends Neuron(dim) {
  def fp(x: Seq[Double]) = x
  def bp(x: Seq[Double], t: Seq[Double]) = x.zip(t).map(loss.tupled)
}
```

中間層も実装する。引数は、中間層が受け取る値の要素数と、活性化関数と、加重を生成する関数と、後続の層である。

```
class Hidden(dim: Int, act: Act, weight: () => SGD, next: Neuron) extends Neuron(dim) {
  lazy val w = List.fill(next.dim, dim)(weight())
  def fp(x: Seq[Double]) = next.fp(act.fp(wx(x)))
  def wx(x: Seq[Double]) = w.map(_.map(_ * _).sum)
  def bp(x: Seq[Double], t: Seq[Double]) = ((z: Seq[Double]) => {
    val bp = next.bp(act.fp(z), t).zip(act.bp(z)).map(_ * _)
    for((w, g) <- w.zip(bp); (sgd, x) <- w.zip(x)) sgd(x * g)
    w.transpose.map(_.zip(bp).map(_ * _).sum)
  })(wx(x))
}
```

最後に、定数項を実装する特殊な中間層も実装する。仕組みは単純で、中間層が受け取る値に定数 1 の要素を追加する。

```
class Offset(dim: Int, act: Act, weight: () => SGD, next: Neuron) extends Neuron(dim) {
  lazy val body = new Hidden(dim + 1, act, weight, next)
  def fp(x: Seq[Double]) = body.fp(x.padTo(dim + 1, 1d))
  def bp(x: Seq[Double], t: Seq[Double]) = body.bp(x.padTo(dim + 1, 1d), t).init
}
```

2.3 ソフトマックス関数

多クラス分類の問題では、最終層の活性化関数に式 (2.4) のソフトマックス関数とし、各クラスの確率分布 p を学習する。誤差関数には、式 (2.9) の**交差エントロピー**を使う。式 (2.9) の $H(p)$ は、確率分布 p の不偏性を表す**平均情報量**である。

$$E_{CE}(p, \hat{p}) = - \int p(\mathbf{y}) \log \hat{p}(\mathbf{y}) d\mathbf{y} = - \int p(\mathbf{y}) \left\{ \log p(\mathbf{y}) - \log \frac{p(\mathbf{y})}{\hat{p}(\mathbf{y})} \right\} d\mathbf{y} = H(p) + D(p\| \hat{p}) \geq D(p\| \hat{p}). \quad (2.9)$$

式 (2.10) の $D(p\| \hat{p})$ を**カルバック・ライブラー情報量**と呼ぶ。これは非負で、確率分布 p, \hat{p} が等価な場合に限り 0 になる。式 (2.10) から確率分布 \hat{p} の項を抽出すると、分布 \hat{p} の対数の期待値である。また、分布 \hat{p} は各層の加重 W_n の尤度である。

$$D(p\| \hat{p}) = \int_K p(y) \log \frac{p(y)}{\hat{p}(y)} dy \geq \int_K p(y) \left(1 - \frac{\hat{p}(y)}{p(y)} \right) dy = 0. \quad (2.10)$$

分類器が推定する確率分布 \hat{p} を真の確率分布 p に近付けるには、式 (2.9) を最小化し、間接的に式 (2.10) を最小化する。これは、尤度 \hat{p} の対数の期待値の最大化に相当し、即ち最尤推定である。例えば、最終層では式 (2.11) の勾配法を行う。

$$\frac{\partial E_{CE}}{\partial z^k} = -\frac{\partial}{\partial z^k} \sum_{i=1}^K y^i \left(\log e^{z^i} - \log \sum_{j=1}^K e^{z^j} \right) = -y^k + \sum_{i=1}^K y^i \hat{y}^k = -y^k + \hat{y}^k. \quad (2.11)$$

以上の議論を踏まえ、式 (2.4) の順伝播と式 (2.11) の勾配計算を実装する。この活性化関数は、最終層でのみ使用できる。

```
class Softmax extends Act {
    def fp(z: Seq[Double]) = z.map(math.exp(_)/z.map(math.exp).sum)
    def bp(z: Seq[Double]) = Seq.fill(z.size)(1.0)
}
```

勾配 $\nabla f(z)$ の計算を誤魔化したので、中間層で使うと、逆伝播が妨害される。その点に目を瞑れば、簡単に実装できた。

```
val model = new Offset(3, new Softmax, ()=>new PlainSGD, new Output(4, _-_))
```

Fig. 2.3 は、国際信号旗の Z 旗を学習する例である。問題が簡単なので、单層の方が多層よりも正確な Z 旗を学習できる。

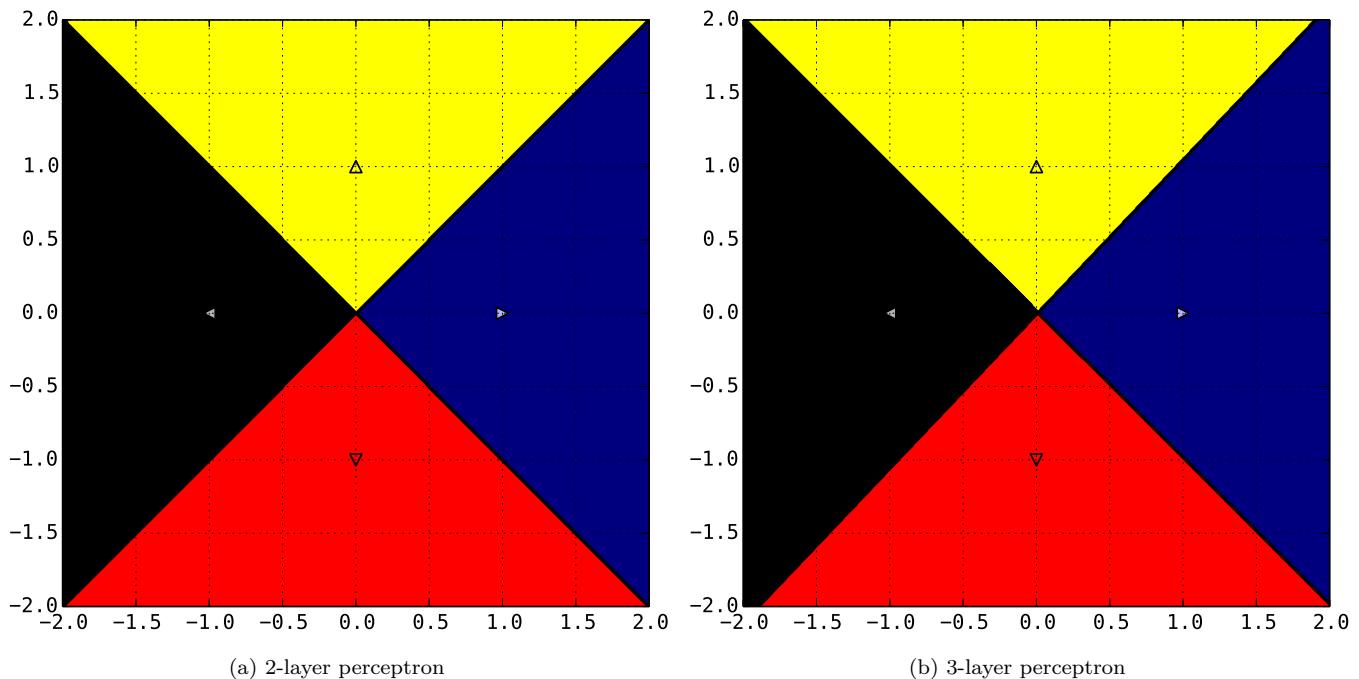


Fig. 2.3: maritime signal flag *zulu* learned by a perceptron.

層を増やすと、表現能力は高まるが、第 2.4 節でも述べる通り、学習が停滞しやすく、却って精度が低下する場合がある。

2.4 鞍点と学習率の調整

勾配法には、最適解に到達する前に鞍点で最適化が停滞する場合がある。式 (2.12) に示す関数 E の最小化の例で考える。鞍点とは、ある方向では極大値だが、別の方向では極小値となる停留点である。関数 E の場合は、原点 \mathbf{O} が鞍点である。

$$\Delta E = \frac{\partial f}{\partial x} \Delta x + \frac{\partial f}{\partial y} \Delta y = 2x\Delta x - 2y\Delta y, \text{ where } E(x, y) = x^2 - y^2. \quad (2.12)$$

原点 \mathbf{O} に嵌ると、Fig. 2.4(a) のように最適化が止まる。しかし、 $y \neq 0$ に動けば、勾配が負になり、最適化を再開できる。鞍点は頻繁に現れる。Fig. 2.4(b) は、5通りの初期値で排他的論理和を学習した際の誤差 E_{sq} の推移で、停滞が見られる。

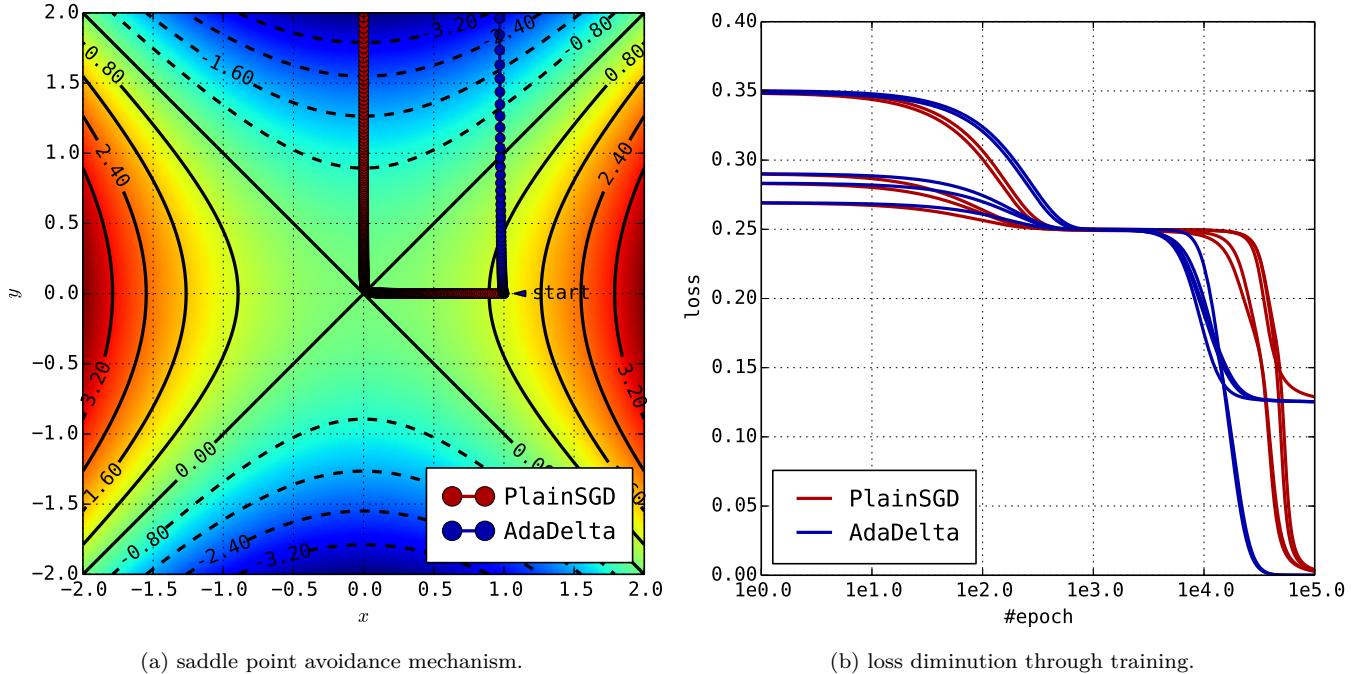


Fig. 2.4: comparison of PlainSGD and AdaDelta.

対策として、最適解の近傍では学習率を小さく、鞍点の近傍で学習率を大きく調節する、適応的な勾配法が利用される。例えば、式 (2.13) の *AdaGrad* は、時刻 t での学習率を勾配 ∇E_t の期待値の逆数とし、また、時刻 t に従って減衰させる。

$$\Delta w = -\frac{\eta}{t\sqrt{\mathbf{E}[(\nabla E)^2]_t}}, \text{ where } \begin{cases} \mathbf{E}[(\nabla E)^2]_t = \frac{1}{t} \sum_{\tau=0}^t (\nabla E_\tau)^2, \\ \mathbf{E}[(\nabla E)^2]_0 = \varepsilon. \end{cases} \quad (2.13)$$

AdaGrad は全時間の勾配を考慮するが、式 (2.14) の *AdaDelta* では、期待値に加重 ρ を導入し、直近の勾配を重視する。式 (2.14) の分数には、加重 w と勾配 ∇E の単位を変換する役割がある。なお、定数 ε はゼロ除算を防ぐ微小な値である。

$$\Delta w_{mt} = -\frac{\sqrt{\mathbf{E}[(\Delta w)^2]_t + \varepsilon}}{\sqrt{\mathbf{E}[(\nabla E)^2]_t + \varepsilon}} \nabla E_{mt}, \text{ where } \begin{cases} \mathbf{E}[x]_t = \rho \mathbf{E}[x]_{t-1} + (1 - \rho)x_t, \\ \mathbf{E}[x]_0 = 0. \end{cases} \quad (2.14)$$

以上の議論を踏まえ、*AdaDelta* を実装する。引数は、定数 ρ, ε である。Fig. 2.4 に、単純な勾配法との性能の比較を示す。

```
class AdaDelta(r: Double = 0.95, e: Double = 1e-8) extends SGD {
    var eW, eE = 0.0
    def apply(dE: Double) = {
        lazy val v = math.sqrt(eW + e) / math.sqrt(eE + e)
        this.eE = r * eE + (1 - r) * math.pow(1 * dE, 2.0)
        this.eW = r * eW + (1 - r) * math.pow(v * dE, 2.0)
        this.w -= v * dE
    }
}
```

2.5 通時的誤差逆伝播法

自然言語や信号など、時系列の未来を予測するには、状態を記憶し、時系列を逐次的に順伝播させる機能が必要である。再帰型ニューラルネットワークが代表的で、中間層の状態 y を、次の時刻の入力 x の順伝播に合流させ、中間層に戻す。

$$y^t = f(z^t) = f(W_i x^t + W_h y^{t-1}). \quad (2.15)$$

再帰構造を展開して、擬似的に再帰を除去すれば、従来通り勾配法で最適化できる。これを**通時的誤差逆伝播法**と呼ぶ。以下に実装を示す。順伝播は、同じ中間層を繰り返し通過し、最後に、後続の層に伝播する。逆伝播も、同様に実装する。

```
class RNN(dim: Int, hidden: Neuron, output: Neuron, value: Double = 0) extends Neuron(dim) {
    val hist = Seq[Seq[Double]]().toBuffer
    val loop = Seq[Seq[Double]]().toBuffer
    def fp(x: Seq[Double]) = output.fp(hp(x).last)
    def tt(x: Seq[Double]) = hist.zip(loop).map(_++_).foldRight(x)
    def hp(x: Seq[Double]) = loop.append(hidden.fp(hist.append(x).last++loop.last))
    def bp(x: Seq[Double], t: Seq[Double]) = tt(output.bp(hp(x).last, t))(hidden.bp)
    def init = hist.clear -> loop.clear -> loop.append(Seq.fill(hidden.dim)(value))
}
```

順伝播や逆伝播を行う度に、中間層の状態が記憶される。従って、時系列の処理が終わる度に、初期化する必要がある。勾配法も実装し直す。逆伝播の間に、中間層の挙動が変わると最適化に障るので、逆伝播の完了まで、加重を据え置く。

```
class DelaySGD(sgd: SGD = new PlainSGD, var d: Double = 0) extends SGD {
    def apply(dE: Double) = (sgd.w = w, sgd(dE), d += sgd.w - w)
    def force = (w += d, d = 0)
}
```

使用例を示す。なお、中間層で逆伝播を繰り返す際に、勾配は引数で渡される。その勾配を、専用の最終層で取り出す。

```
val params = Seq[DelaySGD]().toBuffer
val model3 = new Offset(5, new Sigmoid, ()=>params.append(new DelaySGD).last, new Output(1))
val model2 = new Offset(6, new Sigmoid, ()=>params.append(new DelaySGD).last, new Output(5, (x,e)=>e))
val model1 = new RNN(1, model2, model3)
```

時系列の具体的な例として、波形を学習させる。余弦波を受け取り、位相を遅らせて正弦波を予測する回帰問題である。

```
val x = Seq.tabulate(200)(n => Seq(0.5 * math.cos(0.02 * math.Pi * n) + 0.5))
val y = Seq.tabulate(200)(n => Seq(0.5 * math.sin(0.02 * math.Pi * n) + 0.5))
for(step <- 1 to 100000) model1.init -> x.zip(y).foreach(model1.bp(_,_) -> params.foreach(_.force))
```

Fig. 2.5 に予測した波形と真の波形を示す。学習の初期段階では、歪な波形だが、学習が進むと、綺麗な正弦波に近づく。

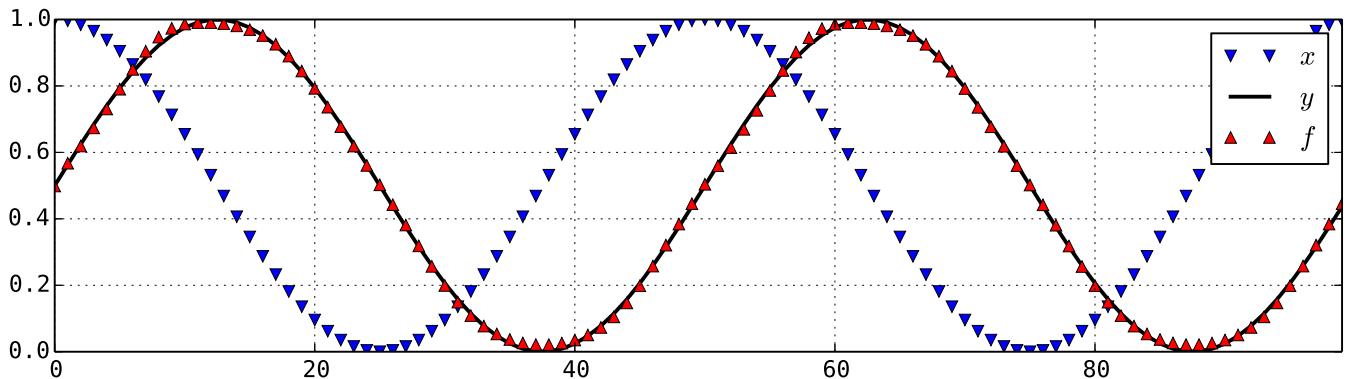


Fig. 2.5: sine curve prediction.

時系列の機械学習は、機械翻訳や映像生成など、発展が顕著な分野で、複雑な文脈構造を如何に捉えるかが課題である。

第3章 サポートベクターマシン

サポートベクターマシンは、分類問題に対し、各クラスの集団からの距離 d が最大になる境界を学習する分類器である。Fig. 3.1(a) に線型分離可能な問題の、(b) に線型分離が困難な問題の例を示す。まずは、線型分離可能な場合を解説する。

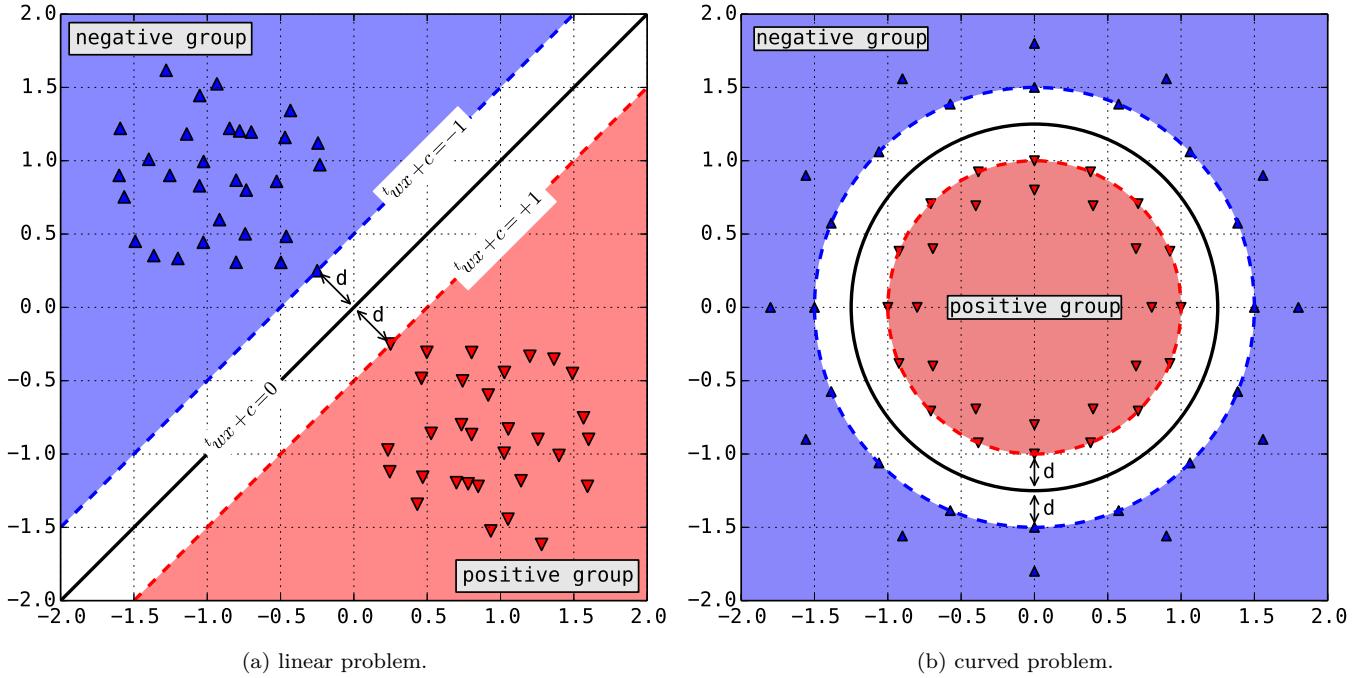


Fig. 3.1: a support vector machine.

Fig. 3.1(a) の分類器は、式 (3.1) に従って、説明変数 x に対し、目的変数 y を推定する。 w は加重で、 c は定数項である。

$$\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x} + c) \in \{1, -1\}. \quad (3.1)$$

距離 d の最適化は**制約付き最適化問題**であり、学習対象の集合 \mathbb{T} に対して、式 (3.2) に示す制約条件を満たす必要がある。

$$\forall (\mathbf{x}, y) \in \mathbb{T}: y(\mathbf{w} \cdot \mathbf{x} + c) \geq 1, \text{ where } y \in \{1, -1\}. \quad (3.2)$$

距離 d は式 (3.3) で求まる。式 (3.2) を念頭に、式を簡略化すると、距離 d の最大化は加重 w の最小化と等価だと言える。

$$d(\mathbb{T}) = \min \frac{|\mathbf{w} \cdot \mathbf{x} + c|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}, \text{ where } \mathbf{x} \in \mathbb{T}. \quad (3.3)$$

現実には、式 (3.2) の**ハードマージン**は、誤分類に対して過剰に敏感なので、式 (3.4) に示す**ソフトマージン**を利用する。

$$\forall (\mathbf{x}, y) \in \mathbb{T}: y(\mathbf{w} \cdot \mathbf{x} + c) \geq 1 - \xi, \text{ where } \xi = \begin{cases} 0 & \text{if } y(\mathbf{w} \cdot \mathbf{x} + c) > 1, \\ |y - (\mathbf{w} \cdot \mathbf{x} + c)| & \text{if } y(\mathbf{w} \cdot \mathbf{x} + c) \leq 1. \end{cases} \quad (3.4)$$

式 (3.4) は、誤分類された点 \mathbf{x} に対し、罰を与える役割がある。 ξ を**ヒンジ損失**と呼ぶ。最終的に式 (3.5) を最小化する。

$$f(\mathbf{w}) = C \sum_n \xi_n + \frac{1}{2} \|\mathbf{w}\|^2, \text{ where } C > 0. \quad (3.5)$$

定数 C は、誤分類の許容量を決定する。小さな値に設定すると誤分類に鈍感になり、大きな値に設定すると敏感になる。

3.1 双対問題の導出

式 (3.5) は、式 (3.4) を束縛条件として、**ラグランジュの未定乗数法**で最小化できる。条件が 2 個ある点に注意を要する。

$$L(\mathbf{w}, c, \xi, \lambda, \mu, \mathbb{T}) = f(\mathbf{w}) - \sum_{i=1}^N \lambda_i \{y_i(\mathbf{w} \cdot \mathbf{x}_i + c) - 1 + \xi_i\} - \sum_{i=1}^N \mu_i \xi_i. \quad (3.6)$$

式 (3.4) の条件は不等式なので、式 (3.7) の**カルーシュ・クーン・タッカー条件**を満たす場合のみ、未定乗数法が使える。

$$\lambda_i \{y_i(\mathbf{w} \cdot \mathbf{x}_i + c) - 1 + \xi_i\} = 0, \begin{cases} \lambda_i \geq 0, \\ \mu_i \geq 0, \\ \mu_i \xi_i = 0. \end{cases} \quad (3.7)$$

変数 λ_i, μ_i は未定乗数である。式 (3.6) を加重 \mathbf{w} と定数 c と未定乗数で偏微分すれば、 L が極値になる条件が得られる。

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial c} = \frac{\partial L}{\partial \lambda} = \frac{\partial L}{\partial \mu} = 0, \Rightarrow \begin{cases} \mathbf{w} = \sum_{i=1}^N \lambda_i y_i \mathbf{x}_i, \\ 0 = \sum_{i=1}^N \lambda_i y_i, \\ \lambda_i = C - \mu_i. \end{cases} \quad (3.8)$$

Fig. 3.1(a) を振り返ると、 $C = 0$ の場合は、式 (3.7) より、境界から距離 d の点だけが $\lambda > 0$ となり、加重 \mathbf{w} に寄与する。その点を**サポートベクトル**と呼ぶ。式 (3.6) に式 (3.8) を代入すると、都合よく ξ や C が消去され、式 (3.9) が得られる。

$$\tilde{L}(\lambda) = \min_{\mathbf{w}, c} L(\mathbf{w}, c, \lambda) = \sum_{i=1}^N \lambda_i \left\{ 1 - \frac{1}{2} \sum_{j=1}^N \lambda_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \right\} \leq f(\mathbf{w}). \quad (3.9)$$

式 (3.9) の \tilde{L} の最大化を $f(\mathbf{w})$ の**ラグランジュ双対問題**と呼ぶ。 \tilde{L} と $f(\mathbf{w})$ の最適値は合致する。これを**強双対性**と呼ぶ。

3.2 逐次的な最適化

式 (3.9) の解析的な最適化は困難なため、**逐次最小問題最適化法**での最適化を検討する。まず、適当な 2 点 $\mathbf{x}_i, \mathbf{x}_j$ を選ぶ。その 2 点の乗数 λ_i, λ_j を式 (3.10) を満たす範囲で最適化する。以上の操作を、全ての点が式 (3.7) を満たすまで繰り返す。

$$y_i \delta_i + y_j \delta_j = 0, \text{ where } \begin{cases} \delta_i = \hat{\lambda}_i - \lambda_i \\ \delta_j = \hat{\lambda}_j - \lambda_j \end{cases} \Leftarrow 0 = \sum_{i=1}^N \lambda_i y_i. \quad (3.10)$$

2 点 $\mathbf{x}_i, \mathbf{x}_j$ に対し、 \tilde{L} の極大値を求める。式 (3.10) に注意して、式 (3.9) を δ_i, δ_j で偏微分すると、式 (3.11) が得られる。

$$\frac{\partial \tilde{L}}{\partial \delta_i} = y_i(y_i - y_j) - \delta_i |\mathbf{x}_i - \mathbf{x}_j|^2 - y_i \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \cdot (\mathbf{x}_i - \mathbf{x}_j). \quad (3.11)$$

乗数 λ_i, λ_j の移動量は式 (3.12) となる。ただし、式 (3.7) を満たす必要があり、 $0 \leq \lambda \leq C$ の範囲で**クリッピング**を行う。

$$\delta_i = -\frac{y_i}{|\mathbf{x}_i - \mathbf{x}_j|^2} \left\{ \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \cdot (\mathbf{x}_i - \mathbf{x}_j) - y_i + y_j \right\}. \quad (3.12)$$

なお、定数 c の値は、 $y(\mathbf{w} \cdot \mathbf{x})$ を最小化する点 \mathbf{x} に着目すると、式 (3.13) で計算できる。以上で、必要な数式が出揃った。

$$c = -\frac{1}{2} \left\{ \min_{i|y_i=+1} \sum_{j=1}^N \lambda_j y_j \mathbf{x}_i \cdot \mathbf{x}_j + \max_{j|y_j=-1} \sum_{i=1}^N \lambda_i y_i \mathbf{x}_j \cdot \mathbf{x}_i \right\}. \quad (3.13)$$

逐次最小問題最適化法の最悪計算時間は $O(n^3)$ だが、点 \mathbf{x}_i を選ぶ際に、式 (3.7) に反する点を重視すると効率的である。

3.3 線型分離の学習

第3.2節までの議論に基づき、逐次最小問題最適化法を実装する。まず、組 (x, y) を実装する。乗数 λ を変数として持つ。

```
case class Data(x: Seq[Double], t: Int, var l: Double = 0) {
  def kkt(svm: SVM, C: Double) = t * svm(this) match {
    case e if e < 1 => l >= C
    case e if e > 1 => l == 0
    case _ => true
  }
}
```

次に、サポートベクターマシンの本体を実装する。引数 k は内積である。敢えて抽象化したのは、第3.4節の布石である。

```
class SVM(data: Seq[Data], k: (Data, Data) => Double) {
  var const = 0.0
  def group(t: Int) = data.filter(_.t == t).map(apply)
  def apply(x: Data) = data.map(d => d.l * d.t * k(x, d)).sum + const
}
```

最後に、逐次最小問題最適化法を実装する。第3.2節に述べた数式を実装し、式(3.7)を満たすまで逐次的に最適化する。

```
class SMO(data: Seq[Data], k: (Data, Data) => Double, C: Double = 1e-10) extends SVM(data, k) {
  while(data.filterNot(_.kkt(this, C)).size >= 2) {
    val a = data(util.Random.nextInt(data.size))
    val b = data(util.Random.nextInt(data.size))
    val min = math.max(-a.l, if(a.t == b.t) b.l - this.C else -b.l)
    val max = math.min(-a.l, if(a.t == b.t) b.l - this.C else -b.l) + C
    val prod = this(Data(a.x.zip(b.x).map(_-_), 0)) - this.const
    val best = -a.t * (prod - a.t + b.t) / (k(a, a) - 2 * k(a, b) + k(b, b))
    if(!best.isNaN) a.l += a.t * math.max(min, math.min(max, best))
    if(!best.isNaN) b.l -= a.t * b.t * math.max(min, math.min(max, best))
    this.const = -0.5 * (group(+1).min + group(-1).max) + this.const
  }
}
```

Fig. 3.2 に学習の例を示す。綺麗な境界を学習できた。(b)では、誤分類によりサポートベクトルが消える様子がわかる。

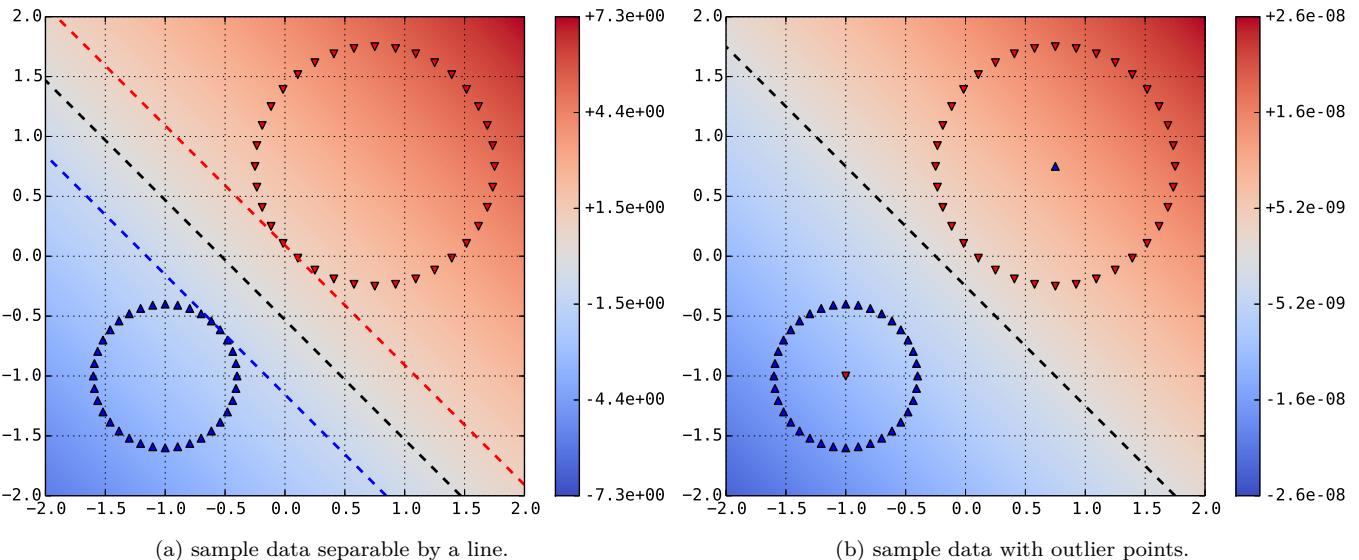


Fig. 3.2: decision surface learned by a linear SVM.

黒の点線はクラスの境界を表し、赤と青の点線はサポートベクトルを表す。赤と青の濃淡は $w \cdot x + c$ の値の勾配を表す。

3.4 特徴空間の変換

第3.2節までの議論は、線型分離可能な問題が前提だった。第3.4節では、線型分離が困難な問題に議論の対象を拡げる。線型分離が困難な問題でも、非線型の適当な関数 Φ で他の空間に写像し、線型分離可能な問題に変換できる場合がある。

$$\Phi : \mathbf{x} \mapsto \Phi_{\mathbf{x}}. \quad (3.14)$$

具体例を挙げると、式 (3.15) の写像 Φ_g は、点 \mathbf{x} を無限の次元を持つ点 $\Phi_{\mathbf{x}}$ に変換する、無限次元の空間への写像である。低次元の空間では、点 \mathbf{x} を線型分離するのが困難でも、無限次元に引き延ばせば、必ず適当な超平面で線型分離できる。

$$\Phi_g(\mathbf{x}) = \exp\left(-\frac{1}{2\sigma^2} \|\mathbf{x}\|^2\right) \left[\frac{1}{\sqrt{n!}} \frac{x_d^n}{\sigma^n} \right]_{dn}, \text{ where } n = 0, 1, 2, \dots, \infty. \quad (3.15)$$

第3.2節までの議論を振り返ると、内積 $\mathbf{x}_i \cdot \mathbf{x}_j$ が何度も現れた。第3.4節では写像 Φ を通すので、 $\Phi_{\mathbf{x}_i} \cdot \Phi_{\mathbf{x}_j}$ の形になる。無限次元の内積の計算量は無限で、写像 Φ の計算も困難である。しかし、**テイラー級数**を使えば、簡単に内積が求まる。

$$\Phi_{\mathbf{x}_i} \cdot \Phi_{\mathbf{x}_j} = \exp\left\{-\frac{1}{2\sigma^2} \|\mathbf{x}_i - \mathbf{x}_j\|^2\right\} \Leftarrow e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \quad (3.16)$$

内積を計算可能な写像 Φ を使うことで、陽に Φ を計算せずに、仮想的な高次元空間に写像する技法を**カーネル法**と呼ぶ。理論的には、**正定値性**を満たす対称関数 k に対し、内積が k で定義される**再生核ヒルベルト空間**への写像 Φ が存在する。

$$k : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{R}, \text{ where } k(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_j, \mathbf{x}_i). \quad (3.17)$$

関数 k が正定値性を満たすとは、点 \mathbf{x} を元に持つ空間 Ω に対し、式 (3.18) の**グラム行列**が正定値行列である場合を指す。

$$\begin{bmatrix} k(\mathbf{x}_i, \mathbf{x}_j) \end{bmatrix}_{ij}, \text{ where } \mathbf{x}_i, \mathbf{x}_j \in \Omega. \quad (3.18)$$

関数 k を利用して空間 Ω を空間 H_k に写像すると、空間 H_k の元 f, g の内積 $\langle f, g \rangle$ は、**再生性**により関数 k で定義できる。

$$\forall a_i, b_j \in \mathbb{R}: \langle f, g \rangle = \left\langle \sum_{i=1}^N a_i k(\mathbf{x}, \mathbf{x}_i), \sum_{j=1}^N b_j k(\mathbf{x}, \mathbf{x}_j) \right\rangle = \sum_{i=1}^N \sum_{j=1}^N a_i b_j k(\mathbf{x}_i, \mathbf{x}_j). \quad (3.19)$$

要するに、正定値性を満たす任意の対称関数 k に対し、内積が関数 k で定義された空間 H_k が存在し、内積を計算できる。最も汎用的な例は、式 (3.16) の**ガウシアンカーネル**である。Fig. 3.3 に、線型分離が困難な問題を学習した結果を示す。

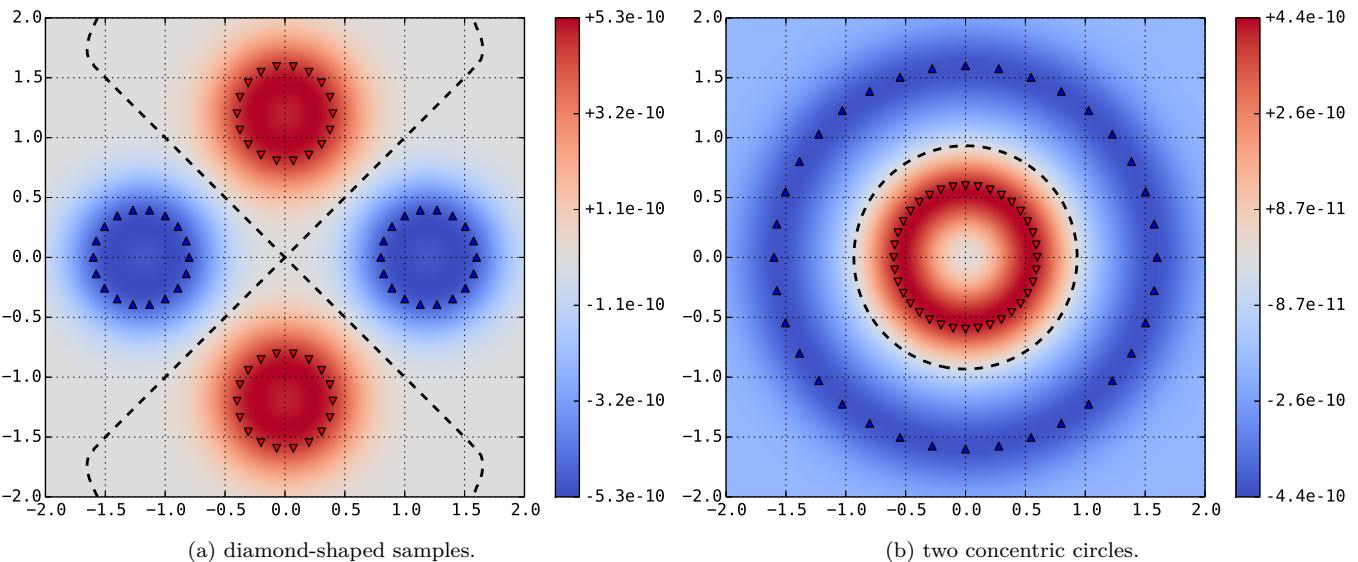


Fig. 3.3: decision surface learned by a Gaussian SVM.

第3.2節に掲載した実装に、適当な内積の定義を与えれば、任意の写像を試せる。手作りで、無限次元の魔法を味わおう。

3.5 ヒルベルト空間

第3.4節は、内積と距離が式(3.17)の関数 k で定義され、無限級数の極限も計算可能な空間 H_k が存在する点に依拠する。空間 H が線型空間で、式(3.20)を満たす関数 $\langle \mathbf{x}, \mathbf{y} \rangle$ が存在する場合に、これを内積と呼び、空間 H を**内積空間**と呼ぶ。

$$\forall a_i, b_j \in \mathbb{R}: \left\langle \sum_{i=1}^I a_i \mathbf{x}_i, \sum_{j=1}^J b_j \mathbf{y}_j \right\rangle = \left\langle \sum_{j=1}^J b_j \mathbf{y}_j, \sum_{i=1}^I a_i \mathbf{x}_i \right\rangle = \sum_{i=1}^I \sum_{j=1}^J a_i b_j \langle \mathbf{x}_i, \mathbf{y}_j \rangle, \langle \mathbf{x}, \mathbf{x} \rangle \geq 0. \quad (3.20)$$

高校数学で学ぶ標準内積は、この定義に従う。また、関数 f, g を元とする空間では、その内積は式(3.21)で定義できる。

$$\langle f, g \rangle = \int_H f(\mathbf{x}) \overline{g(\mathbf{x})} d\mu(\mathbf{x}). \quad (3.21)$$

関数 μ は関数空間 H の**測度**である。さて、内積空間 H では、式(3.22)に示す通り、内積を使って**ノルム**を定義できる。

$$\|x\| = \langle x, x \rangle^{\frac{1}{2}} \in \mathbb{R}. \quad (3.22)$$

式(3.22)を利用して、任意の2点の距離 d を定義できる。距離 d が式(1.2)を満たす場合に、空間 H は**距離空間**である。

$$d(x, y) = \|x - y\| \geq 0. \quad (3.23)$$

空間 H で定義された任意の級数が、空間 H の元に収束する場合に、空間 H は**完備性**を満たし、**ヒルベルト空間**となる。正定値性を満たす適当な対称関数 k を定義して、関数 k の線型結合で式(3.24)の空間 H_0 を作る。これを**線型包**と呼ぶ。

$$H_0 = \text{span} \left\{ \sum_{n=1}^N a_n k(\mathbf{x}_n, \cdot) \mid a_n \in \mathbb{R} \right\}. \quad (3.24)$$

空間 H_0 の元 f, g に対し、内積を式(3.25)の通りに定義する。証明は省くが、空間 H_0 はヒルベルト空間の条件を満たす。

$$\langle f, g \rangle_{H_0} = \sum_{i=1}^I \sum_{j=1}^J a_i b_j k(\mathbf{x}_i, \mathbf{x}_j), \text{ where } \begin{cases} f(\mathbf{x}) = \sum_{i=1}^I a_i k(\mathbf{x}_i, \mathbf{x}), \\ g(\mathbf{x}) = \sum_{j=1}^J b_j k(\mathbf{x}_j, \mathbf{x}). \end{cases} \quad (3.25)$$

ぜひ、式(3.25)が式(3.20)の性質を満たし、その内積で距離 d を定義すると、式(1.2)の公理を満たす点を確認しよう。さて、式(3.25)より自明だが、空間 H_0 の元 f は、式(3.26)の再生性を満たし、空間 H_0 は**再生核ヒルベルト空間**となる。

$$f(\mathbf{x}) = \sum_{n=1}^N a_n k(\mathbf{x}_n, \mathbf{x}) = \langle f, k(\cdot, \mathbf{x}) \rangle_{H_0}. \quad (3.26)$$

再生性を持つ関数 k を**再生核**と呼ぶ。核は式(3.27)に示す**積分変換**に由来する。これは、空間 Ω_s, Ω_t の間の写像である。

$$F(s) = \int_{\Omega_t} k(s, t) f(t) dt, \text{ where } \begin{cases} s \in \Omega_s, \\ t \in \Omega_t. \end{cases} \quad (3.27)$$

例えば、**ラプラス変換**や**フーリエ変換**が該当する。さて、式(3.28)で定義される核関数 k は、再生核である。証明しよう。

$$k(x, y) = \frac{a}{\pi} \text{sinc } a(y - x). \quad (3.28)$$

式(3.21)に従って内積を求めるか、式(3.29)を得る。式(3.28)が矩形関数の双対である点に注目して、再生性を導ける。

$$\langle f, k(x, \cdot) \rangle_{L^2} = \int_{-\infty}^{\infty} f(y) \overline{k(x, y)} dy = \frac{1}{2\pi} \int_{-a}^a \mathcal{F}_f(\omega) e^{i\omega x} d\omega = f(x). \quad (3.29)$$

積分変換と機械学習の関係は興味深く、特に、深層学習の優れた性能の理由を積分変換による研究は、注目に値する。簡単な例では、式(3.30)に示す**シグモイドカーネル**の挙動は、式(2.1)で加重 w が固定されたニューロンと等価になる。

$$k(w, x) = \tanh^t w x. \quad (3.30)$$

深層学習は、勾配法を通じて加重 w を最適化するため、自在に最適化される高次元空間の層を持つこと等価だと言える。

第4章 決定木の学習と汎化性能

意思決定の分野では、しばしば**決定木**と呼ばれる、質問と条件分岐の再帰的な木構造で、条件 x と結論 y の関係を表す。例えば、式 (4.1) は、海水浴の是非 y を判断する決定木である。気象 x に対し、質問と条件分岐を繰り返し、結論を導く。

$$y \approx f(\mathbf{x}) = \begin{cases} 0 & \text{if } \text{wavy}(\mathbf{x}) = 1 \\ \text{otherwise} \begin{cases} 0 & \text{if } \text{rain}(\mathbf{x}) = 1 \\ 1 & \text{if } \text{rain}(\mathbf{x}) = 0 \end{cases} & \text{if } \text{wavy}(\mathbf{x}) = 0 \end{cases} \quad (4.1)$$

決定木の学習では、意思決定の事例の集合 $\{\mathbf{x}, y\}$ に対し、簡潔で解釈の容易な質問と条件分岐と、その順序を習得する。

4.1 情報源符号化定理

理想的な決定木は、簡潔明瞭である。即ち、最低限の質問で、結論に至る。ここで、**情報源符号化**の概念を導入しよう。質問と条件分岐を繰り返す過程は、条件 x の情報を分解し、情報の断片に2進数の**符号語**を割り振る操作と等価である。

$$C(\mathbf{x}): \mathbf{x} \mapsto s \in \{0, 10, 11\}. \quad (4.2)$$

事象 \mathbf{x} が従う確率分布 $p(\mathbf{x})$ を仮定して、事象 \mathbf{x} が伴う情報の価値 $I(\mathbf{x})$ を定義する。質問の妥当性に相当する量である。情報の価値とは、その事象の希少性である。即ち、価値 $I(\mathbf{x})$ は、確率 $p(\mathbf{x})$ に対して単調減少であり、式 (4.3) を満たす。

$$p(\mathbf{x}_i) \leq p(\mathbf{x}_j) \Leftrightarrow I(\mathbf{x}_i) \geq I(\mathbf{x}_j). \quad (4.3)$$

また、複数の事象が同時に発生した場合の情報の価値は、個別に発生した場合の情報の価値の総和となると自然である。

$$I(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = \sum_{n=1}^N I(\mathbf{x}_n). \quad (4.4)$$

式 (4.4) の性質を**情報の加法性**と呼ぶ。以上の性質を満たす定義を考えると、式 (4.5)を得る。この $I(\mathbf{x})$ を**情報量**と呼ぶ。

$$I(\mathbf{x}) = \log_2 \frac{1}{p(\mathbf{x})} = -\log_2 p(\mathbf{x}) \geq 0. \quad (4.5)$$

符号 C の符号語を圧縮するには、事象 \mathbf{x} に、情報量に応じた長さの符号語を割り振る。これを**エントロピー符号**と呼ぶ。符号語の長さの期待値 \bar{L} は、式 (4.6) の**シャノンの情報源符号化定理**に従う。関数 H を確率分布 p の**平均情報量**と呼ぶ。

$$\bar{L}(C) \geq H(p) = \sum_{\mathbf{x}} p(\mathbf{x}) I(\mathbf{x}) = -\sum_{\mathbf{x}} p(\mathbf{x}) \log_2 p(\mathbf{x}) \geq 0. \quad (4.6)$$

情報量の議論を利用して、質問の回数を圧縮する方法を検討しよう。まず、最初の質問は、情報量が最大の質問を選ぶ。質問 Q により、集合 X が K 通りの部分集合 X_k に分割される場合は、質問 Q の情報量 $G(Q)$ は、式 (4.7) で定義される。

$$G(Q) = H(X) - H(X|Q) = H(X) - \sum_{k=0}^{K-1} P(X_k | X) H(X_k) \geq 0. \quad (4.7)$$

同様に、部分集合 X_k に対し、情報量が最大の質問を選び、次の質問とする。この操作を繰り返し、最適な決定木を得る。質問の回数は整数なので、決定木が表す分布 \hat{p} は、分布 p と異なる。その様子を表す $H(p, \hat{p})$ を**交差エントロピー**と呼ぶ。

$$\bar{L}(C) = H(p, q) = -\sum_{\mathbf{x}} p(\mathbf{x}) \log \hat{p}(\mathbf{x}) = -\sum_{\mathbf{x}} p(\mathbf{x}) \left\{ \log p(\mathbf{x}) - \log \frac{p(\mathbf{x})}{\hat{p}(\mathbf{x})} \right\} = H(p) + D(p||q) \geq H(p). \quad (4.8)$$

余分な質問の回数を表す $D(p||\hat{p})$ を**カルバック・ライブラーア情報量**と呼ぶ。これは、確率分布 p, \hat{p} の差を表す量でもある。

4.2 条件分岐の最適化

第4.1節の議論に基づき、決定木を実装する。まず、推論を抽象化する。条件 x は整数の列で、結論 y は任意の型とする。

```
trait Node[T] extends (Seq[Int] => T)
```

次に、決定木の本体を実装する。引数は、決定木が学習する集合 $\{x, y\}$ と、決定木の末端の細分化を抑える閾値である。決定木は再帰的に生成されるが、質問の情報量が微小の場合は、分布 $p(y)$ の最大値を与える y を定数値として出力する。

```
case class Question[T](x: Seq[(Seq[Int], T)], limit: Double = 1e-5) extends Node[T] {
    lazy val m = x.groupBy(_._2).maxBy(_._2.size)._1
    lazy val p = x.groupBy(_._2).map(_._2.size.toDouble / x.size)
    lazy val ent = p.map(p => -p * math.log(p)).sum / math.log(2)
    lazy val division = x.head._1.indices.map(split).minBy(_._ent)
    def apply(x: Seq[Int]) = if(ent - division.ent < limit) m else division(x)
    def split(v: Int) = x.map(_._1(v)).toSet.map(Division(x, v, _)).minBy(_._ent)
}
```

次に、条件分岐を実装する。引数は、決定木が学習する集合 $\{x, y\}$ と、条件 x を分割する軸と、分割を行う閾値である。分割する軸と値は、式 (4.7) で議論した通り、質問の情報量を最大化する軸と値が選択される。これで決定木が完成した。

```
case class Division[T](x: Seq[(Seq[Int], T)], axis: Int, value: Int) extends Node[T] {
    val sn1 = Question(x.filter(_._1(axis) > value))
    val sn2 = Question(x.filter(_._1(axis) <= value))
    val ent = (sn1.ent * sn1.x.size + sn2.ent * sn2.x.size) / x.size
    def apply(x: Seq[Int]) = if(x(axis) >= value) sn1(x) else sn2(x)
}
```

Fig. 4.1は、各々が正規分布に従う3クラスの点の集合を学習し、決定木で空間をそれらのクラスに分割した結果である。結果的に、過剰に複雑な境界となった。個別の事例には忠実だが、正規分布の形からは乖離した。これを過学習と呼ぶ。

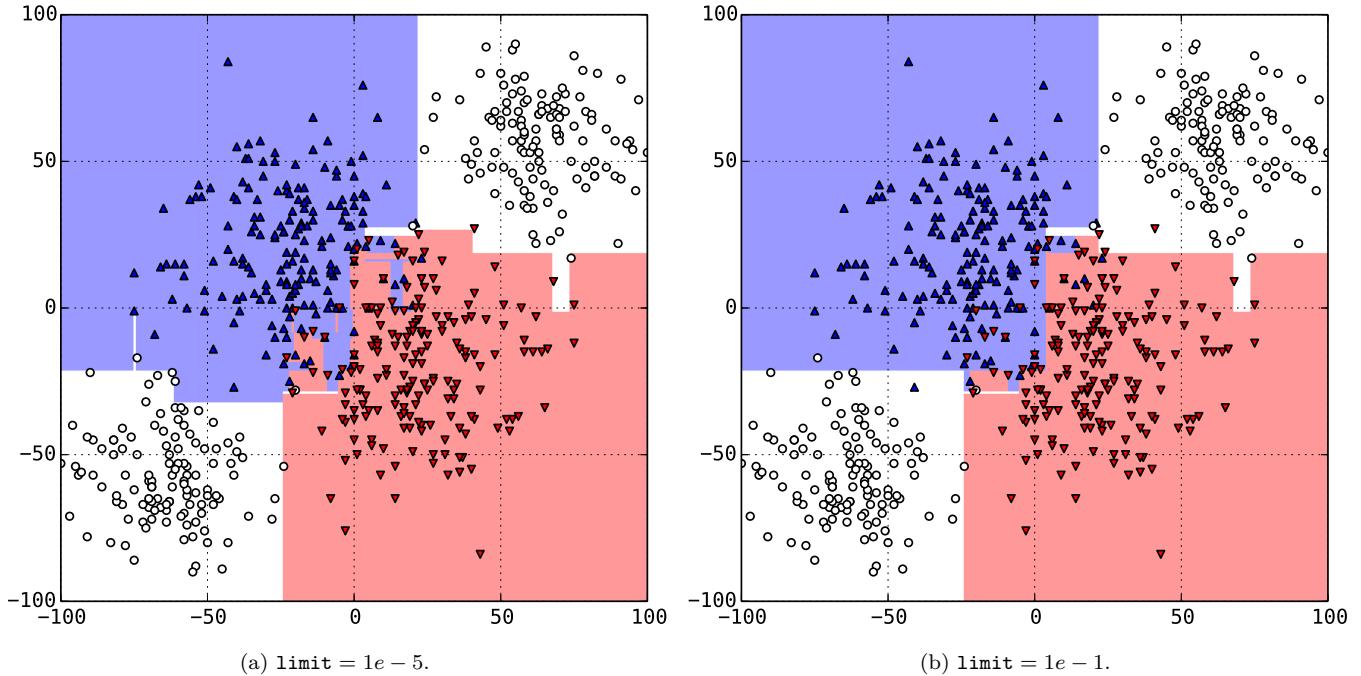


Fig. 4.1: region segmentation by a decision tree.

過学習は、機械学習で普遍的な課題だが、特に、決定木は、際限なく細分化できるため、しばしば表現能力が過剰である。過学習を抑えるには、Fig. 4.1(b) のように、分割の閾値を調節するか、第4.3節で学ぶアンサンブル学習が有効である。

4.3 アンサンブル学習

決定木に限らず、機械学習では、真の関係 f と、習得した関係 \hat{f} の間に若干の差があり、それが過学習として顕在化する。過学習の原因は、教師データの偏りや、関数 \hat{f} の過剰な表現能力にある。関数 f, \hat{f} の差を 2 乗誤差関数で定式化しよう。

$$\int_{\mathbf{x}} p(\mathbf{x}) (y - \hat{f}(\mathbf{x}))^2 d\mathbf{x} = \mathbf{V}[y - f(\mathbf{x})] + (\mathbf{E}[\hat{f}(\mathbf{x})] - f(\mathbf{x}))^2 + \mathbf{V}[\hat{f}(\mathbf{x})]. \quad (4.9)$$

式 (4.9) の第 2 項の平方根を **バイアス** と、第 3 項を **バリアンス** と呼ぶ。この 2 項が過学習や、逆に学習不足の原因となる。両者はトレードオフの関係にあるが、 T 個の関数 \hat{f}_t を **弱学習器** とし、投票を行う **アンサンブル学習** により、調節できる。

$$\hat{f}(\mathbf{x}) = \frac{1}{T} \sum_{t=1}^T \hat{f}_t(\mathbf{x}). \quad (4.10)$$

過学習を防ぐには、相互に独立した弱学習器の訓練が重要である。そこで、事例を選び直す **ブートストラップ法** を行う。要素の重複を許して T 通りの部分集合を作成し、 T 通りの弱学習器 f_t を訓練する。これで、式 (4.11) の分散が低減する。

$$\mathbf{V}[\hat{f}(\mathbf{x})] = \frac{1}{T^2} \sum_{i=1}^T \sum_{j=1}^T \mathbf{C}[f_i(\mathbf{x}), f_j(\mathbf{x})]. \quad (4.11)$$

この手法は **バギング** とも呼ばれる。基本的には、決定木や深層学習など表現能力が過剰な手法を使う。以下に実装する。

```
case class Bagging[T](x: Seq[(Seq[Int], T)], t: Int, n: Int) extends Node[T] {
  val f = Seq.fill(t)(Question(Seq.fill(n)(x(util.Random.nextInt(x.size)))))
  def apply(x: Seq[Int]) = f.map(_(x)).groupBy(identity).maxBy(_.size)._1
}
```

4.4 ブースティング法

逆に、学習不足の解消には、弱学習器の表現能力を補う弱学習器を作り、加重投票を行う **ブースティング法** を適用する。

$$\hat{f}(\mathbf{x}) = \sum_{t=1}^T w_t \hat{f}_t(\mathbf{x}). \quad (4.12)$$

弱学習器 f_t は、弱学習器 f_1, \dots, f_{t-1} が判断を誤った点 \mathbf{x} を重点的に学習する。点 \mathbf{x} を選ぶ確率分布 $q_t(\mathbf{x})$ を検討しよう。分類問題を想定し、式 (4.13) に示す **指數誤差** を最小化する。指數誤差の最小化は、関数 f, \hat{f} の値の内積の最大化である。

$$E(\hat{f}) = \exp \left\{ -\mathbf{f}(\mathbf{x}) \hat{f}(\mathbf{x}) \right\} = \exp \left\{ -\mathbf{f}(\mathbf{x}) \sum_{t=1}^T w_t \hat{f}_t(\mathbf{x}) \right\} \geq 0. \quad (4.13)$$

次に、特に分類問題を想定し、関数 \hat{f} が取り得る値に制約条件を設定する。関数 f, \hat{f} の値は、式 (4.14) の条件を満たす。

$$\sum_{k=1}^K f(\mathbf{x}, k) = \sum_{k=1}^K \hat{f}(\mathbf{x}, k) = 1, \text{ where } \begin{cases} \mathbf{f}(\mathbf{x}) = [f(\mathbf{x}, k)]_k, \\ \hat{f}(\mathbf{x}) = [\hat{f}(\mathbf{x}, k)]_k. \end{cases} \quad (4.14)$$

具体的には、真の関数 f は、式 (4.15) に示す値を取る。ただし、関数 \hat{f} は、式 (4.14) を満たす範囲で、自由な値を取る。

$$f(\mathbf{x}, k) = \begin{cases} 1 & \text{if } y = k, \\ \frac{1}{1-K} & \text{if } y \neq k. \end{cases} \quad (4.15)$$

式 (4.13) を分解すると、式 (4.16) を得る。この関数 q_T を確率分布として、弱学習器 f_T が学習する集合を無作為に選ぶ。

$$E(\hat{f}) = q_T(\mathbf{x}) \exp \left\{ -\mathbf{f}(\mathbf{x}) w_T \hat{f}_T(\mathbf{x}) \right\}, \text{ where } q_T(\mathbf{x}) = \exp \left\{ -\mathbf{f}(\mathbf{x}) \sum_{t=1}^{T-1} w_t \hat{f}_t(\mathbf{x}) \right\}. \quad (4.16)$$

弱学習器 f_T に対し、指數誤差 E を最小化する加重 w_T は、式 (4.17) で計算できる。**ラグランジュの未定乗数法** を使った。

$$\hat{w}_T = \log \left(\frac{1 - E_T}{E_T} \right) + \log(K - 1), \text{ where } E_T = q_T(\mathbf{x}, y) \mathbb{I}(\hat{f}_T(\mathbf{x}) \neq y). \quad (4.17)$$

以上の手法は、*AdaBoost* と呼ばれる。他にも、弱学習器の誤差を別の弱学習器で補う、**勾配ブースティング** が存在する。

4.5 汎化性能の最適化

第4.3節の議論を踏まえ、AdaBoostを実装しよう。基本的には、弱学習器を逐次的に作成し、追加する操作を繰り返す。

```
case class AdaBoost[T](x: Seq[(Seq[Int], T)], m: Int) extends Node[T] {
  val k = x.map(_._2).toSet
  val t = Seq(AdaStage(x, Seq.fill(x.size)(1.0 / x.size), m)).toBuffer
  def apply(x: Seq[Int]) = k.maxBy(y => t.map(_.score(x, y)).sum)
  while(t.last.best.error < 0.5) t += AdaStage(x, t.last.next, m)
}
```

弱学習器 f_t は、 M 個の候補 \hat{f}_{tm} を作成し、最も高精度な候補を f_t とする。この操作を E_t が 0.5 を超えるまで繰り返す。

```
case class AdaStage[T](x: Seq[(Seq[Int], T)], p: Seq[Double], m: Int) extends Node[T] {
  val best = List.fill(m)(Resample(x, p.map(_ / p.sum))).minBy(_.error)
  val gain = math.log((1 / best.error - 1) * (x.map(_._2).toSet.size - 1))
  val next = x.map(score).map(gain - _).map(math.exp).zip(p).map(_ * _)
  def score(x: Seq[Int], y: T) = if(this(x) == y) gain else 0
  def apply(x: Seq[Int]) = best(x)
}
```

弱学習器の候補 \hat{f}_{tm} は、確率分布 q_t に従う部分集合 \mathbb{T}_t を学習する。集合 \mathbb{T}_t はノイマンの棄却法で擬似的に抽出できる。

```
case class Resample[T](x: Seq[(Seq[Int], T)], p: Seq[Double]) extends Node[T] {
  val data = Seq[(Seq[Int], T)]().toBuffer
  def reject(i: Int) = if(util.Random.nextDouble * p.max < p(i)) x(i) else null
  while(data.size < p.size) data += reject(util.Random.nextInt(p.size)) -= null
  def error = x.map((x, y) => this(x) != y).zip(p).filter(_._1).map(_._2).sum
  def apply(x: Seq[Int]) = quest(x)
  val quest = Question(data.toList)
}
```

Fig. 4.2 は、各々が正規分布に従う 3 クラスの事例を学習した結果である。比較のため、Fig. 4.1 と同じ事例を使用した。

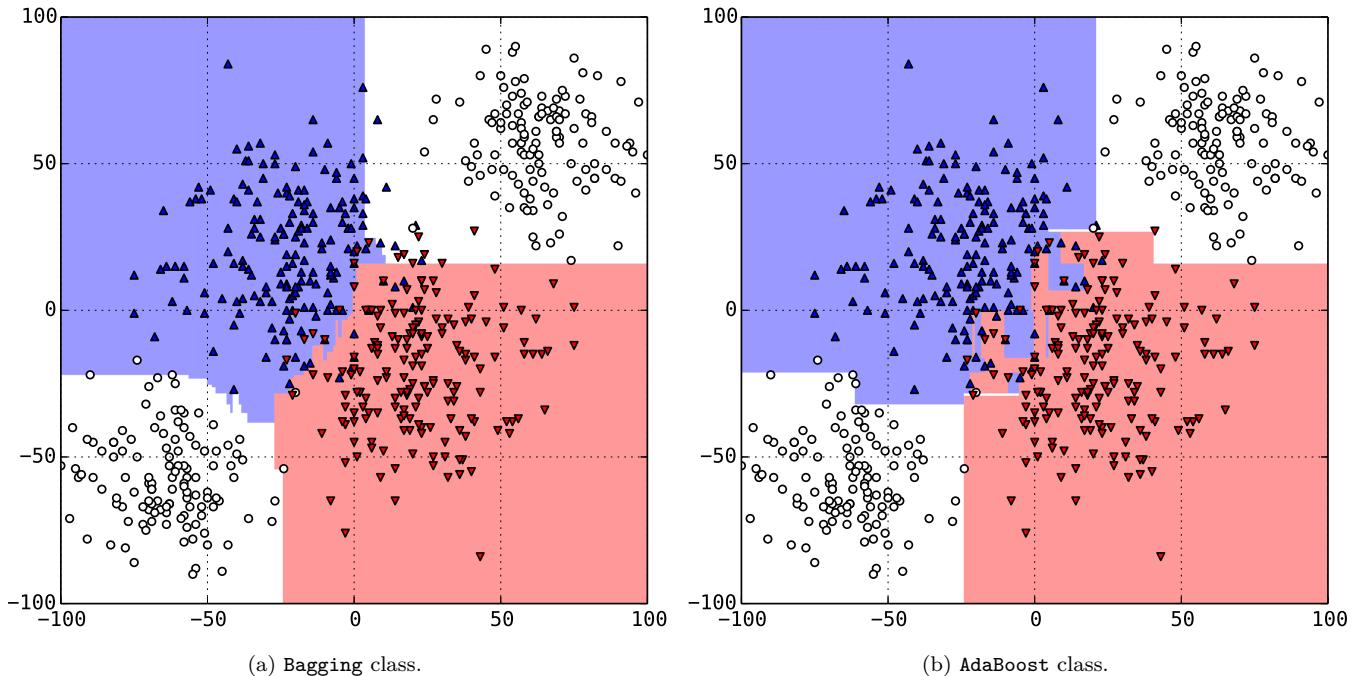


Fig. 4.2: region segmentation by ensemble learning.

決定木は、表現能力が過剰なので、学習不足を補うブースティングより、過学習を抑えるバギングの方が効果的である。

4.6 圧縮アルゴリズム

第4.1節で学んだ情報源符号化は、機械学習よりも、可逆圧縮の理論として知られる。その代表例が**ハフマン符号**である。決定木に似た**ハフマン木**を作り、その分岐に2進数の符号語を割り当て、再帰的に圧縮と復元を行う。以下に実装を示す。

```
abstract class Node(val s: String, val w: Int) {
  def decode(bits: String, root: Node = this): String
  def encode(text: String, root: Node = this): String
}
```

引数は、この頂点が表す文字と、文字が現れる頻度である。また、文字列を受け取り、圧縮と復元を行う機能を実装する。以下に、末端の頂点を実装する。復元の際は、頂点に対応する文字を出力し、圧縮の際は、何もせず最上位の頂点に戻る。

```
case class Atom(ch: String, freq: Int) extends Node(ch, freq) {
  def decode(bits: String, root: Node) = if(bits.isEmpty) ch else ch ++ root.decode(bits, root)
  def encode(text: String, root: Node) = if(text.size < 2) "" else root.encode(text.tail, root)
}
```

次に、符号語を表す頂点を実装する。厳密には、頂点と符号語の1桁を紐付けた頂点で、圧縮の際に、その桁を出力する。

```
case class Code(node: Node, bit: String) extends Node(node.s, node.w) {
  def decode(bits: String, root: Node) = node.decode(bits.tail, root)
  def encode(text: String, root: Node) = bit++node.encode(text, root)
}
```

分岐も実装する。復元の際は、符号語の0,1に対応する頂点を、圧縮の際は、圧縮する文字を含む頂点を選び、巡回する。

```
case class Fork(nodes: Seq[Code]) extends Node(nodes.map(_.s).mkString, nodes.map(_.w).sum) {
  def decode(bits: String, root: Node) = nodes.find(_.bit.head == bits.head).get.decode(bits, root)
  def encode(text: String, root: Node) = nodes.find(_.s.contains(text.head)).get.encode(text, root)
}
```

次に、再帰的に木構造を構築する手順を実装する。まず、頻度が最低の部分木の組を選び、その親となる分岐を構築する。また、部分木の頻度を合計し、新たな部分木の頻度とする。この操作を逐次的に繰り返し、完全なハフマン木を構築する。

```
implicit class Huffman(nodes: Seq[Node]) {
  def fork: Seq[Code] = nodes.zipWithIndex.map(_ -> _.toString).map(Code(_, _))
  def join: Seq[Node] = Seq(Fork(nodes.take(2).fork)).union(nodes.tail.tail)
  def tree: Seq[Node] = if(nodes.size <= 1) nodes else join.sortBy(_.w).tree
}
```

最後に、暗黙の型変換を利用して、文字列からハフマン木を生成する機能も実装した。この文字列が、圧縮の対象となる。

```
implicit class Symbols(source: String) {
  def countFreq = source.split("").groupBy(identity).mapValues(_.size)
  def toHuffman = countFreq.toSeq.map(Atom(_, _)).sortBy(_.w).tree.head
}
```

以上で、可逆圧縮が完成した。以下に、使用例を示す。なお、未知の文字を圧縮すると、例外が発生する点に、注意する。

```
val encoded = "Lorem ipsum dolor sit amet consectetur adipiscing elit".toHuffman.encode("lorem")
val decoded = "Lorem ipsum dolor sit amet consectetur adipiscing elit".toHuffman.decode(encoded)
println(encoded)
println(decoded)
```

以下に、出力を示す。文字が出現する頻度の偏りに起因して、平均情報量が抑制されたため、半分の圧縮率を達成できた。

```
110011110010100011111
lorem
```

第5章 潜在的ディリクレ配分法

自然言語の機械学習には、特有の困難がある。特定の形態素が出現する確率は低く、その確率も話題に応じて変化する。話題も曖昧で多岐に渡り、教師あり学習が困難なので、単語 w の背後にある話題 z を、教師なし学習する方法を考える。

5.1 確率的潜在意味解析

話題 z は観測できず、潜在的な情報である。また、話題 z の分布は、記事の主題に応じて変化する。その点を考慮しよう。具体的には、単語 w の出現が、試行 1 回の**多項分布**に従うと考える。また、単語 w と話題 z が従う確率分布を仮定する。

$$P(w, z | \phi, \theta) = P(w | \phi) P(\phi) P(z | \theta) P(\theta) = \left(\prod_{v=1}^V \phi_{zv}^{N_v} \right) \text{Dir}(\phi | \nu) \left(\prod_{k=1}^K \theta_k^{N_k} \right) \text{Dir}(\theta | \alpha). \quad (5.1)$$

変数 N_v, N_k は、単語 v と話題 k の出現の数で、総和は 1 である。変数 ϕ_v, θ_k は、単語 v と話題 k が出現する確率である。式 (5.2) の多項分布は、話題 k が確率 θ_k で現れる記事から N 語を取得して、話題 k の単語が N_k 個となる確率を与える。

$$P(z | \theta) = N! \prod_{k=1}^K \frac{\theta_k^{N_k}}{N_k!}, \text{ where } \sum_{k=1}^K N_k = N. \quad (5.2)$$

式 (5.3) の**ディリクレ分布**は、話題 k の単語が $N_k - 1$ 個だった場合に、実際に話題 k が確率 θ_k で出現する確率を与える。これは、変数 N を連続量に拡張した多項分布である。式 (5.3) に従う話題 z の推定を、**潜在的ディリクレ配分法**と呼ぶ。

$$P(\theta) = \text{Dir}(\theta | N) = \Gamma\left(\sum_{k=1}^K N_k\right) \prod_{k=1}^K \frac{\theta_k^{N_k-1}}{\Gamma(N_k)} = \frac{1}{B(N)} \prod_{k=1}^K \theta_k^{N_k-1}. \quad (5.3)$$

式 (5.3) で、関数 Γ は**ガンマ関数**で、自然数の階乗 $(n-1)!$ を複素数の階乗に拡張した関数である。式 (5.4) に定義する。

$$\Gamma(n) = \int_0^\infty x^{n-1} e^{-x} dx. \quad (5.4)$$

関数 B は**ベータ関数**を多変量に拡張した複素関数で、式 (5.2) に現れる多項係数の逆数に相当する。式 (5.5) が成立する。

$$B(N) = \int \prod_{k=1}^K x_k^{N_k+1} dx = \int \cdots \int \prod_{k=1}^K x_k^{N_k+1} dx_1 dx_2 \cdots dx_K, \text{ where } \sum_{k=1}^K x_k = 1. \quad (5.5)$$

式 (5.5) から、式 (5.6) が簡単に導ける。式 (5.6) の性質は、確率 θ を実際の記事から推定する際に、重要な役割を果たす。

$$P(z) = \int P(z | \theta) P(\theta) d\theta = N! \frac{B(\hat{\alpha})}{B(\alpha)} \prod_{k=1}^K \frac{1}{N_k!}, \text{ where } \hat{\alpha}_k = \alpha_k + N_k. \quad (5.6)$$

記事を学習すると、確率 θ_k の最適値は式 (5.7) に従う。これを事後確率と呼ぶ。また、式 (5.2) を確率 θ_k の尤度と呼ぶ。学習前では、どの話題の出現も均等と仮定し、式 (5.3) に従って、確率 θ_k に初期値を設定できる。これを事前確率と呼ぶ。

$$\theta_k \sim P(\theta | z) = \frac{P(z | \theta) P(\theta)}{P(z)} = \frac{1}{B(\hat{\alpha})} \prod_{k=1}^K \theta_k^{\hat{\alpha}_k-1}. \quad (5.7)$$

式 (5.7) は、観測を重視して、尤度を最適化する最尤推定と対照的で、観測の偏りを重視する。これを**ベイズ推定**と呼ぶ。最尤推定では、観測の偏りに起因した過学習が発生するが、その点が解消される。さて、式 (5.5) から式 (5.8) が導ける。

$$\mathbf{E}[\theta_k] \mathbf{E}[\phi_{kv}] = \frac{\hat{\alpha}_k}{\|\hat{\alpha}\|_1} \frac{\hat{\nu}_{kv}}{\|\hat{\nu}_k\|_1}, \text{ where } \|\hat{\alpha}\|_1 = \sum_{k=1}^K \alpha_k, \|\hat{\nu}_k\|_1 = \sum_{v=1}^V \nu_{kv}. \quad (5.8)$$

式 (5.8) に従う乱数により、変数 z を何度も選び直すと、最終的に真の分布 θ に収束する。これを**モンテカルロ法**と呼ぶ。第6章で学ぶ変分ベイズ法と比較して、収束に時間を要するが、複雑な確率分布にも適用でき、並列処理も容易である。

5.2 潜在的な話題の学習

第5.1節の潜在的ディリクレ配分法を実装する。まず、単語と話題の組 (w, z) を実装する。話題 z は無作為に初期化する。

```
case class Word[W](v: W, k: Int) {
  var z = util.Random.nextInt(k)
}
```

潜在的ディリクレ配分法の本体を実装する。引数は、記事と単語の集合に、話題の総数と、母数 α, ν の初期値を与える。

```
class LDA[D,W](texts: Map[D,Seq[W]], val k: Int, a: Double = 0.1, n: Double = 0.01) {
  val words = texts.map(_ -> _.map(Word(_,k)))
  val vocab = words.flatMap(_.2).groupBy(_.v)
  val nd = words.map((d,s) => d -> Array.tabulate(k)(k => s.count(_.z == k) + a)).toMap
  val nv = vocab.map((v,s) => v -> Array.tabulate(k)(k => s.count(_.z == k) + n)).toMap
  val nk = Array.tabulate(k)(k => nv.map(_.2(k)).sum)
  def apply(k: Int) = vocab.keys.toList.filter(v => nv(v).max == nv(v)(k))
  def probs(v: W, d: D) = 0.until(k).map(k => nv(v)(k) * nd(d)(k) / nk(k))
}
```

以上の実装を継承して、モンテカルロ法を実装する。まず、適当な組 (w, z) を選び、その分を変数 α_z, ν_z から除去する。次に、式 (5.8) に従う乱数をノイマンの棄却法で生成し、話題 z を選び直し、母数 α_z, ν_z に加える。この手順を繰り返す。

```
class Gibbs[D,W](texts: Map[D,Seq[W]], k: Int, epochs: Int = 500) extends LDA(texts, k) {
  for(epoch <- 1 to epochs; (document,words) <- util.Random.shuffle(words); w <- words) {
    nk(w.z) -= 1
    nv(w.v)(w.z) -= 1
    nd(document)(w.z) -= 1
    val uni = util.Random.between(0, probs(w.v,document).sum.toDouble)
    w.z = probs(w.v,document).scan(0.0)(_+_).tail.indexWhere(_ >= uni)
    nd(document)(w.z) += 1
    nv(w.v)(w.z) += 1
    nk(w.z) += 1
  }
}
```

以上で完成した。使用例を示す。これは、素数 k を話題と、その倍数を単語と見做し、無作為に生成した数列を学習する。単語 v の共起と、記事毎に異なる話題の分布を再現した。学習が進むと、同じ約数を持つ整数が、同じ話題に分配される。

```
val bases = Seq(2,3,5,7,11)
def sample(n: Int, m: Int, k: Int) = Seq.fill(n)(k * util.Random.nextInt(m / k + 1))
val texts = Seq.fill(1000)(bases.map(sample(util.Random.nextInt(100),50,_)).flatten)
val gibbs = new Gibbs(texts.indices.zip(texts).toMap, bases.size)
```

5.3 単語の類似度の推定

確率 ϕ を単語の意味を表す変数と考え、その距離に従って、単語を分類しよう。第6章で実装する k -means を利用する。

```
val kmeans = new Kmeans(gibbs.nv.values.map(_.toList).toSeq, gibbs.k)
val topics = texts.flatten.distinct.topicBy(v => kmeans(gibbs.nv(v)))
for(topic <- topics.values) println(topic.toSeq.sorted.mkString(","))
```

以下に、出力を示す。共通の約数を持つ自然数が綺麗に分離できた。共起に基づく確率的な話題推定の有効性が窺える。

```
0,7,14,21,28,35,42,49,56,63,70,77
5,10,15,20,25,30,40,45,50,55,60,65,75,80
3,6,9,12,18,24,27,33,36,39,48,51,54,57,66,69,72,78
2,4,8,16,22,26,32,34,38,44,46,52,58,62,64,68,74,76
```

第6章 混合正規分布と最尤推定

適当な観測量 \mathbf{x} から、それが従う確率分布 p を推定する手法が最尤推定である。具体的には、分布 p の母数を推定する。

$$\forall \mathbf{x}: \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{pmatrix} \sim p(\mathbf{x}). \quad (6.1)$$

例えば、正規分布 \mathcal{N} を仮定する場合は、平均 $\boldsymbol{\mu}$ と分散 S が母数に該当する。ただし、分散 S とは分散共分散行列を指す。

$$\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, S) = \frac{1}{\tilde{\mathcal{N}}(S)} \exp \left\{ -\frac{1}{2} {}^t(\mathbf{x} - \boldsymbol{\mu}) S^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right\}, \text{ where } \tilde{\mathcal{N}}(S) = \sqrt{(2\pi)^D |S|}. \quad (6.2)$$

正規分布では簡単なので、複数の正規分布の線型和を考えよう。式 (6.3) を**混合正規分布**と呼ぶ。定数 w_k は加重である。

$$\mathbf{x} \sim p(\mathbf{x}) = \sum_{k=1}^K w_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, S_k), \text{ where } \sum_{k=1}^K w_k = 1. \quad (6.3)$$

正規分布を点 \mathbf{x} の集団または**クラスタ**と見做せば、混合正規分布の最尤推定は、点 \mathbf{x} が属する集団 C_k の推定と同義である。

$$P(\mathbf{x} \in C_k) = \frac{P(C_k) P(\mathbf{x} | C_k)}{P(\mathbf{x})} = \frac{w_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, S_k)}{p(\mathbf{x})}. \quad (6.4)$$

点 \mathbf{x} がどの集団 C_k に属するかは観測できず、潜在的な情報である。この情報を変数 z で表すと、変数 z は潜在変数となる。Fig. 6.1 は、混合正規分布の例である。式 (6.3) の母数を推定し、点 \mathbf{x} で支配的な集団を求めれば、点 \mathbf{x} の帰属がわかる。

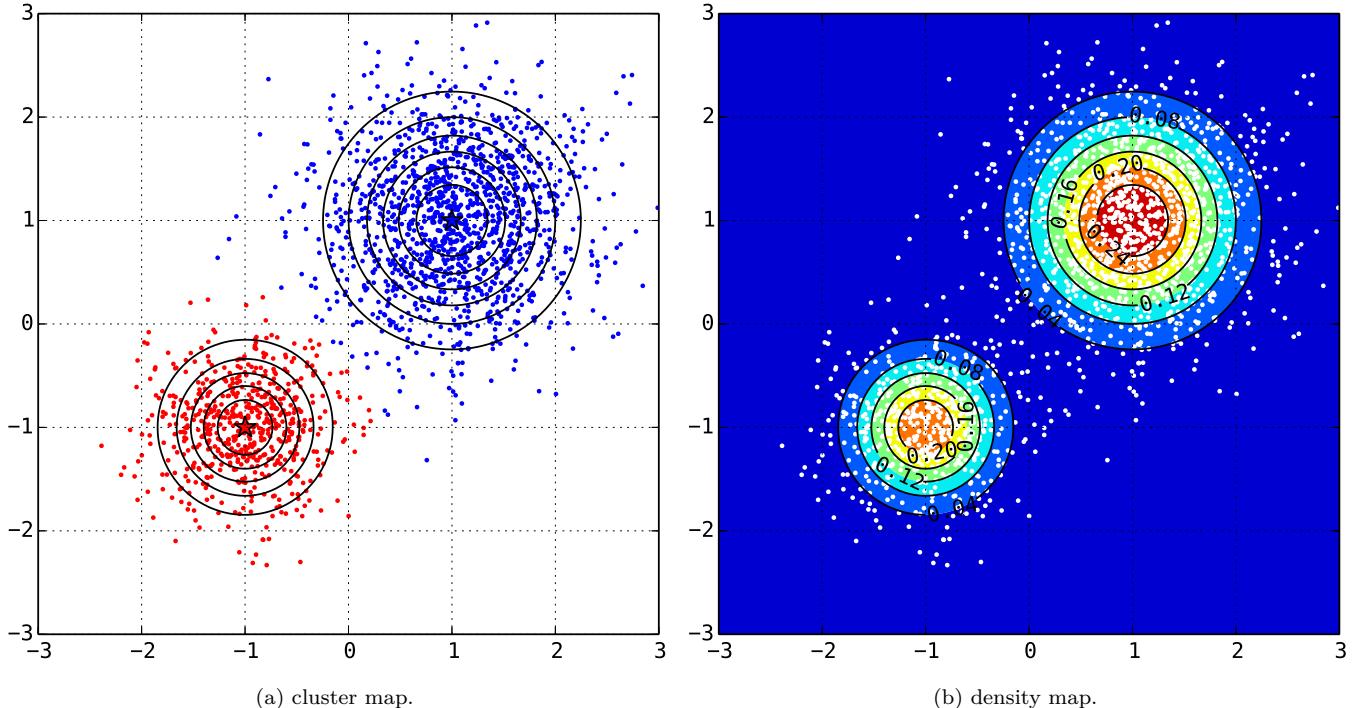


Fig. 6.1: Ground-truth data of a Gaussian mixture model.

Fig. 6.1(a) の分類問題を、**クラスタリング**と呼ぶ。推定対象の値が潜在変数な点を指して、教師なし学習とも呼ばれる。

6.1 クラスタリングの実装

第6.1節では、混合正規分布や最尤推定の議論は忘れて、集合を最適なクラスタに分割する、素朴な方法を検討しよう。理想的な集合 C_k では、その要素 x と、集合 C_k の重心 μ_k の距離が最短となる。この命題を定式化して、式 (6.5) を得る。

$$\min \mathcal{D} = \min \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|x_n - \mu_k\|^2, \text{ where } \hat{z}_{nk} = \begin{cases} 1, & \text{if } x_n \in C_k, \\ 0, & \text{if } x_n \notin C_k. \end{cases} \quad (6.5)$$

式 (6.5) の最適化は、逐次的に行う。まず、重心 μ_k を乱数で初期化する。次に、式 (6.6) に従って、変数 z_{nk} を修正する。

$$\hat{z}_{nk} = \begin{cases} 1, & \text{if } k = \arg \min_j \|x_n - \mu_j\|^2, \\ 0, & \text{if } k \neq \arg \min_j \|x_n - \mu_j\|^2. \end{cases} \quad (6.6)$$

最後に、式 (6.7) により、重心 μ_k を修正する。式 (6.7) は、変数 z_{nk} を固定して、式 (6.5) を重心 μ_k で微分すると導ける。

$$\hat{\mu}_k = \frac{1}{N_k} \sum_{n=1}^N z_{nk} x_n, \text{ where } N_k = \sum_{n=1}^N z_{nk}, \Leftarrow \frac{\partial \mathcal{D}}{\partial \mu_k} = 0. \quad (6.7)$$

以上の手順を繰り返し、最適解を得る。この手法を *k-means* と呼ぶ。混合正規分布を仮定した考察は、第6.2節で行う。以下に実装する。引数は、分割を行う点 x の集合と、分割後に得られる集団 C_k の個数と、操作を繰り返す回数である。

```
class Kmeans(x: Seq[Seq[Double]], k: Int, epochs: Int = 100) {
  val mu = Array.fill(k, x.map(_.size).min)(math.random)
  def apply(x: Seq[Double]) = mu.map(quads(x)(_).sum).zipWithIndex.minBy(_._1)._2
  def quads(a: Seq[Double])(b: Seq[Double]) = a.zip(b).map(_-_).map(d=> d * d)
  def estep = x.groupBy(apply).values.map(c=> c.transpose.map(_.sum / c.size))
  for(epoch <- 1 to epochs) estep.zip(mu).foreach(_.toArray.copyToArray(_))
}
```

Fig. 6.2 は、2 個の正規分布の混合分布に従う Fig. 6.1 の散布図を K 個の集団に分割した様子で、星型の点は重心を表す。

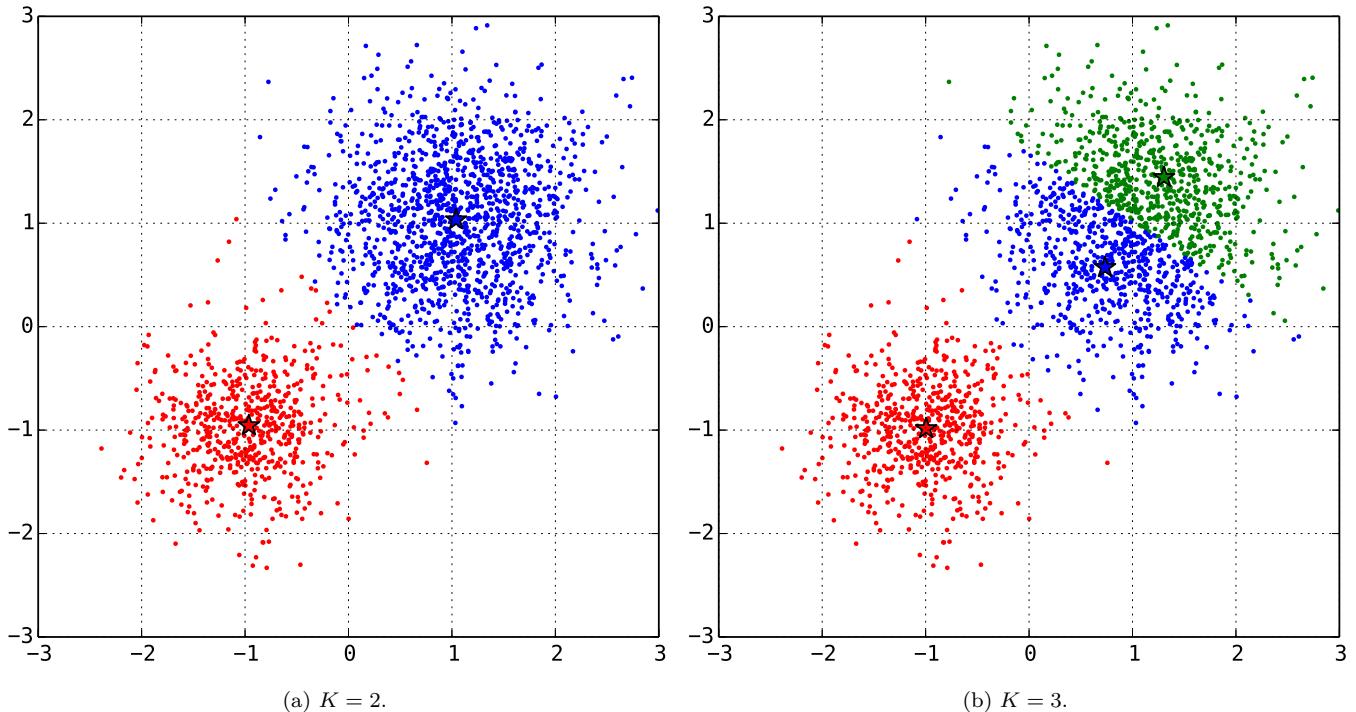


Fig. 6.2: *k-means* clustering on Gaussian mixture model.

なお、正規分布の分散を考慮せず、同じ広がりを持つ集団を想定した点が、課題である。その様子は Fig. 6.2 にも窺える。

6.2 期待値最大化法の理論

第6.2節では、潜在変数 z を、その値が確率的に決まる**確率変数**と考え、分散を含む、混合正規分布の母数を推定しよう。観測変数 \mathbf{x} に対し、潜在変数 z の確率は、式(6.8)で求まる。観測に基づき推定した確率なので、これを事後確率と呼ぶ。

$$P(z_{nk} | \mathbf{x}_n, \theta) = \frac{w_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k)}{p(\mathbf{x}_n)} = \gamma_{nk}. \quad (6.8)$$

次に、混合正規分布の尤度を定義する。尤度 $\mathcal{L}(\theta)$ は母数 θ の妥当性を表し、尤度の最大値を探す操作が最尤推定である。

$$\mathcal{L}(\theta) = P(\mathbf{x} | \theta) = \prod_{n=1}^N \sum_{k=1}^K w_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k). \quad (6.9)$$

微分計算の都合により、尤度を対数化して、対数尤度を最小化する母数を計算しよう。重心 $\boldsymbol{\mu}_k$ による偏微分の例を示す。

$$\frac{\partial}{\partial \boldsymbol{\mu}_k} \log \mathcal{L}(\theta) = \frac{\partial}{\partial \boldsymbol{\mu}_k} \sum_{n=1}^N \log \sum_{k=1}^K w_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k) = \sum_{n=1}^N \gamma_{nk} S_k^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_k). \quad (6.10)$$

加重と重心と分散の推定値 $\hat{w}_k, \hat{\boldsymbol{\mu}}_k, \hat{S}_k$ は式(6.11)となる。加重のみ、式(6.3)より、**ラグランジュの未定乗数法**で求めた。

$$\hat{w}_k = \frac{N_k}{N}, \quad \begin{cases} \hat{\boldsymbol{\mu}}_k = \frac{1}{N_k} \sum_{n=1}^N \gamma_{nk} \mathbf{x}_n, \\ \hat{S}_k = \frac{1}{N_k} \sum_{n=1}^N \gamma_{nk} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^t (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k), \end{cases} \quad \text{where } N_k = \sum_{n=1}^N \gamma_{nk}. \quad (6.11)$$

式(6.11)より、事後確率 γ が求まれば、母数も求まるが、式(6.8)より、事後確率 γ の計算には、母数の値が必要である。従って、解析的な求解は困難である。ここで、凸関数 f と、正の実数 γ_n は、式(6.12)の**イエンゼンの不等式**を満たす。

$$\sum_{n=1}^N \gamma_n f(\mathbf{x}_n) \geq f\left(\sum_{n=1}^N \gamma_n \mathbf{x}_n\right), \quad \text{where } \sum_{n=1}^N \gamma_n = 1. \quad (6.12)$$

対数が凸関数である点に注意して、式(6.12)に式(6.9)を代入して、式(6.13)の関数 Q を得る。これを**補助関数**と呼ぶ。

$$\log \mathcal{L}(\theta) = \max_{\gamma} Q(\gamma, \theta) \geq \sum_{n=1}^N \sum_{k=1}^K \gamma_{nk} \log \frac{w_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k)}{\gamma_{nk}} = Q(\gamma, \theta). \quad (6.13)$$

補助関数 Q は、式(6.14)に示す、変数 γ, θ の修正を交互に繰り返すと単調増加し、最終的に、有限な実数値に収束する。

$$\begin{cases} \hat{\gamma}^{t+1} = \arg \max_{\gamma} Q(\gamma, \theta^t), \\ \hat{\theta}^{t+1} = \arg \max_{\theta} Q(\gamma^t, \theta). \end{cases} \quad (6.14)$$

式(6.14)で、変数 γ^t, θ^t の最適値を求めると、式(6.8)と式(6.11)を得る。両者を交互に修正すると、尤度が最大化する。式(6.8)で変数 γ を修正する操作は、式(6.15)に示す、対数尤度の期待値を計算する操作である。これを**E-step**と呼ぶ。

$$\mathbf{E}_{\mathbf{z}} [\log P(\mathbf{x}, \mathbf{z} | \theta)] = \int_{\mathbf{z}} P(\mathbf{z} | \mathbf{x}, \theta) \log P(\mathbf{x}, \mathbf{z} | \theta) d\mathbf{z} = \sum_{n=1}^N \sum_{k=1}^K \gamma_{nk} \log \{w_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k)\}. \quad (6.15)$$

式(6.11)で変数 θ を修正する操作は、尤度を最大化する。これを**M-step**と呼び、両者を合わせて**期待値最大化法**と呼ぶ。なお、単位行列 E と実数値 λ を使って、分散を λE と置くと、極限 $\lambda \rightarrow 0$ で式(6.16)が成立し、変数 γ_{nk} も z_{nk} になる。

$$\lim_{\lambda \rightarrow 0} \lambda \log \{w \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, \lambda E)\} = \lim_{\lambda \rightarrow 0} \left\{ \lambda \log w - \lambda \frac{D}{2} \log(2\pi\lambda) - \frac{1}{2} \|\mathbf{x} - \boldsymbol{\mu}\|^2 \right\} = -\frac{1}{2} \|\mathbf{x} - \boldsymbol{\mu}\|^2. \quad (6.16)$$

即ち、式(6.17)が成立し、その最大化は式(6.5)の最小化に帰結する。**k-means**は、期待値最大化法の特殊な例と言える。

$$\lim_{\lambda \rightarrow 0} \lambda \mathbf{E}_{\mathbf{z}} [\log P(\mathbf{x}, \mathbf{z} | \theta)] = -\frac{1}{2} \sum_{n=1}^N \sum_{k=1}^K z_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2. \quad (6.17)$$

また、期待値最大化法も、第6.4節で学ぶ変分ベイズ法の特殊な場合であり、第6.2節と酷似した式が、何度も登場する。

6.3 期待値最大化法の実装

第6.2節の議論に基づき、期待値最大化法を実装する。まず、 K 個の D 変量正規分布からなる混合正規分布を実装する。

```
class GMM(val d: Int, val k: Int) {
  val w = Array.fill(k)(1.0 / k)
  val m -> s = (Array.fill(k, d)(math.random), Array.fill(k, d)(math.random))
  def apply(x: Seq[Double]) = w.lazyZip(m).lazyZip(s).map(Normal(x)(_, _, _).p)
}
```

正規分布も実装する。引数は、加重と平均と分散である。なお、分散共分散行列を対角行列と仮定し、実装を単純化した。

```
case class Normal(x: Seq[Double])(w: Double, m: Seq[Double], s: Seq[Double]) {
  def n = math.exp(-0.5 * x.zip(m).map(_ - _).map(d => d * d).zip(s).map(_ / _).sum)
  def p = w * n / math.pow(2 * math.Pi, 0.5 * x.size) / math.sqrt(s.product)
}
```

次に、最適化の手順を実装する。期待値最大化法の E -step と M -step を繰り返す。また、点 x が属す集団 C_k を推定する。

```
class EM(val x: Seq[Seq[Double]], val mm: GMM, epochs: Int = 100) {
  def mstep(P: Seq[Seq[Double]]) = {
    P.map(_.sum / x.size).copyToArray(mm.w)
    val m = P.map(_.zip(x).map((p, x) => x.map(x => p * x)).transpose.map(_.sum))
    val s = P.map(_.zip(x).map((p, x) => x.map(x => p * x * x)).transpose.map(_.sum))
    m.zip(P).map((m, p) => m.map(_ / p.sum)).zip(mm.m).foreach(_.copyToArray(_))
    s.zip(P).map((s, p) => s.map(_ / p.sum)).zip(mm.s).foreach(_.copyToArray(_))
    for ((s, m) <- mm.s.zip(mm.m); d <- 0 until mm.d) s(d) -= m(d) * m(d)
  }
  for (epoch <- 1 to epochs) mstep(x.map(mm(_)).map(p => p.map(_ / p.sum)).transpose)
}
```

Fig. 6.3 は、Fig. 6.1 と同じ散布図を、期待値最大化法で学習した結果で、Fig. 6.1 と同様に、確率密度関数を可視化した。

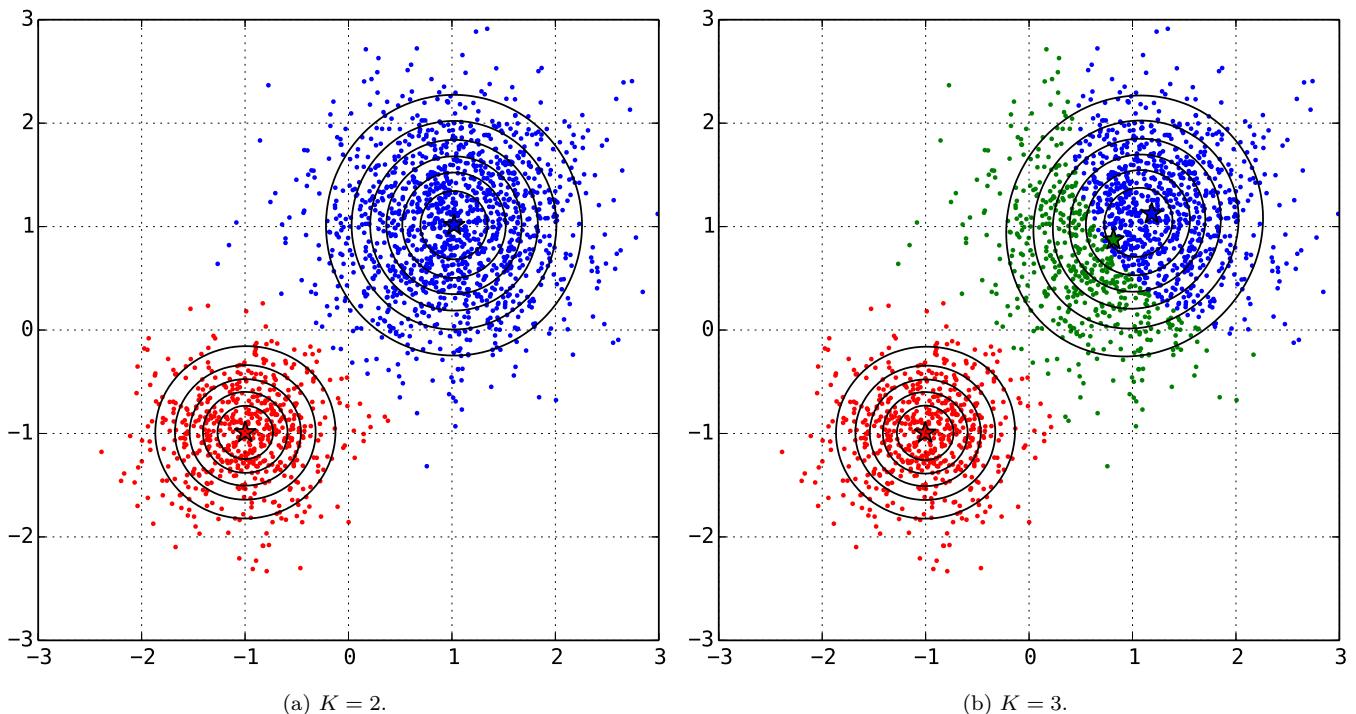


Fig. 6.3: expectation maximization on a Gaussian mixture model.

期待値最大化法では、Fig. 6.2 の k -means と比較して、正規分布の密度の強弱を、境界付近の色分けに正しく反映できる。

6.4 変分ベイズ推定の理論

第6.2節の最尤推定では、母数の最適値を推定した。第6.4節で議論するベイズ推定では、母数の確率分布を推定できる。特に、最適解が複数ある場合にも対応でき、過学習の抑制効果も期待できる。議論を始めるに当たり、尤度を定義しよう。

$$\mathcal{L}(\theta) = p(\mathbf{x} | \theta) = \int p(\mathbf{x}, z) dz = \int p(\mathbf{x} | z) p(z | \theta) dz. \quad (6.18)$$

第6.4節では、潜在変数 z に加え、母数 θ も確率変数に含める。母数 θ の確率分布に母数 ϕ を設定し、尤度を定義し直す。

$$\mathcal{L}(\phi) = p(\mathbf{x} | \phi) = \iint p(\mathbf{x}, z, \theta | \phi) dz d\theta = \iint p(\mathbf{x} | z) p(z | \theta) p(\theta | \phi) dz d\theta. \quad (6.19)$$

式(6.18)に対し、式(6.19)を周辺尤度と呼ぶ。第6.2節と同様に、補助関数 F を定義する。関数 \hat{p} は、適当な分布である。

$$\log \mathcal{L}(\phi) = \log \iint \hat{p}(z, \theta) \frac{p(\mathbf{x}, z, \theta)}{\hat{p}(z, \theta)} dz d\theta \geq \iint \hat{p}(z, \theta) \log \frac{p(\mathbf{x}, z, \theta)}{\hat{p}(z, \theta)} dz d\theta = F(\hat{p}). \quad (6.20)$$

補助関数 F を最大化すると、尤度 \mathcal{L} に収束する。その差は、式(6.21)に示すカルバッカ・ライブラー情報量の形になる。式(6.21)は、変数 z, θ が従う分布 \hat{p} を仮定した場合の、分布 \hat{p}, p の平均情報量の差である。両者が同じ場合に 0 となる。

$$\log \mathcal{L}(\mathbf{x}) - F(\hat{p}) = \iint \hat{p}(z, \theta) \log p(\mathbf{x}) dz d\theta - F(\hat{p}) = \iint \hat{p}(z, \theta) \log \frac{\hat{p}(z, \theta)}{p(z, \theta | \mathbf{x})} dz d\theta = D(\hat{p} \| p) \geq 0. \quad (6.21)$$

関数 \hat{p} を引数に取る関数 F を、汎関数と呼ぶ。汎関数 F の極値を与える引数 \hat{p} を探索する問題は、変分問題と呼ばれる。残念ながら、複数の引数を取る関数 \hat{p} の探索は難しく、式(6.22)に示す平均場近似により、変数間の独立性を仮定する。

$$\hat{p}(z, \theta) = f(z)g(\theta), \text{ where } \begin{cases} \int f(z) dz = 1, \\ \int g(\theta) d\theta = 1. \end{cases} \quad (6.22)$$

関数 f, g に対する汎関数 F の変分問題を解く。ここで、式(6.23)に示すオイラー・ラグランジュ方程式の特殊形を使う。

$$\frac{\partial}{\partial f} \frac{\partial F}{\partial z} = \frac{\partial}{\partial f} \int f(z)g(\theta) \log \frac{p(\mathbf{x}, z, \theta)}{f(z)g(\theta)} d\theta = 0. \quad (6.23)$$

関数 f の値を固定し、単に変数と考えて偏微分すると、式(6.24)を得る。関数 g に対し、式(6.22)の制約条件を使った。

$$\frac{\partial}{\partial f} \int f(z)g(\theta) \log \frac{p(\mathbf{x}, z, \theta)}{f(z)g(\theta)} d\theta = \int g(\theta) \log \frac{p(\mathbf{x}, z, \theta)}{f(z)g(\theta)} d\theta - 1 = 0. \quad (6.24)$$

式(6.24)から、関数 f の最適値を求める。式(6.22)の近似で仮定した、変数 z, θ 間の独立性より、式(6.25)が成立する。

$$\log f(z) = \log f(z) \int g(\theta) d\theta = \int g(\theta) \log f(z) d\theta. \quad (6.25)$$

関数 f, g の最適値 \hat{f}, \hat{g} は、式(6.26)となる。関数 f, g を交互に修正すると、補助関数 F が増加し、周辺尤度に収束する。式(6.26)は、式(6.14)の E -step と M -step に対応し、第6.2節で学んだ期待値最大化法に対し、変分ベイズ法と呼ばれる。

$$\begin{cases} \hat{f}(z) \propto \exp \int g(\theta) \log p(\mathbf{x}, z, \theta) d\theta = \exp \mathbf{E}_g [\log p(\mathbf{x}, z, \theta)], \\ \hat{g}(\theta) \propto \exp \int f(z) \log p(\mathbf{x}, z, \theta) dz = \exp \mathbf{E}_f [\log p(\mathbf{x}, z, \theta)]. \end{cases} \quad (6.26)$$

なお、母数 θ に対し、適当な事前分布を設定すると、分布 \hat{g} と事前分布 p の乖離を抑制し、過学習を防ぐ効果が生じる。

$$F(\hat{p}) = \iint f(z)g(\theta) \log \frac{p(\mathbf{x}, z | \theta)}{f(z)} \frac{p(\theta)}{g(\theta)} dz d\theta = \mathbf{E}_{f,g} \left[\log \frac{p(\mathbf{x}, z | \theta)}{f(z)} \right] - D(g(\theta) \| p(\theta)). \quad (6.27)$$

無限の個数の点 x_n を学習した場合の尤度は、ラプラス近似で式(6.28)と近似でき、ベイズ情報量基準の形が出現する。

$$F(\hat{p}) \simeq \mathbf{E}_{f,g} \left[\log \frac{p(\mathbf{x}, z | \theta)}{f(z)} \right] - \frac{|\theta|}{2} \log N + \log p(\hat{\theta}). \quad (6.28)$$

式(6.28)には、疎な基底を学習し、母数の個数 $|\theta|$ を実質的に削減する正則化の効果があり、過学習の抑制が期待できる。

6.5 母数の事前分布の設定

潜在変数 z や母数 θ の事前分布を注意深く設定すると、事前分布と事後分布が同じ形の分布になり、計算が容易になる。これを**共役分布**と呼ぶ。混合正規分布の母数にも、共役分布が存在する。まず、潜在変数 z が多項分布に従うと仮定する。

$$p(z|w) = \prod_{n=1}^N \prod_{k=1}^K w_k^{z_{nk}}, \text{ where } \forall n: \sum_{k=1}^K z_{nk} = 1. \quad (6.29)$$

式 (6.29) は、潜在変数 z に対する加重 w の尤度でもある。加重 w の事前分布を、式 (6.30) のディリクレ分布で定義する。これは、 K 個の排反事象の反復試行で、事象 k の出現が $\alpha_k - 1$ 回だった場合に、事象 k の確率が w_k である確率を表す。

$$p(w) = \text{Dir}(w|\alpha) = \Gamma\left(\sum_{k=1}^K \alpha_k\right) \prod_{k=1}^K \frac{w_k^{\alpha_k-1}}{\Gamma(\alpha_k)} = \frac{1}{B(\alpha)} \prod_{k=1}^K w_k^{\alpha_k-1}. \quad (6.30)$$

関数 Γ はガンマ関数で、階乗を拡張した複素関数である。関数 B はベータ関数で、多項係数を拡張した複素関数である。平均 μ の事前分布には、式 (6.31) の正規分布を仮定する。母数 σ には、式 (6.31) の分散を徐々に減少させる効果がある。

$$p(\mu|S) = \prod_{k=1}^K \mathcal{N}(\mu_k | m_k, \sigma_k^{-1} S_k). \quad (6.31)$$

式 (6.31) は、平均 μ に対する分散 S の尤度でもある。分散 S の事前分布は、式 (6.32) の**逆ウィシャート分布**を仮定する。

$$p(S) = \prod_{k=1}^K \mathcal{W}(S_k^{-1} | W_k, \nu_k) = \prod_{k=1}^K \frac{1}{\tilde{\mathcal{W}}(W_k, \nu_k)} |S_k^{-1}|^{\frac{\nu_k-D-1}{2}} \exp\left\{-\frac{1}{2}\text{tr}(W_k^{-1} S_k^{-1})\right\}. \quad (6.32)$$

これは、分散 W の D 変量正規分布に従う ν 個の変数 x_n の直積 $x_n^t x_n$ の和の分布である。即ち、標本分散の分布である。

$$\tilde{\mathcal{W}}(W_k, \nu_k) = 2^{\frac{\nu_k D}{2}} \pi^{\frac{D(D-1)}{4}} |W_k|^{\frac{\nu_k}{2}} \prod_{d=0}^{D-1} \Gamma\left(\frac{\nu_k - d}{2}\right). \quad (6.33)$$

Fig. 6.4 は、Fig. 6.1 と同じ散布図を、変分ベイズ法で学習した結果で、Fig. 6.1 と同様に、確率密度関数を可視化した。

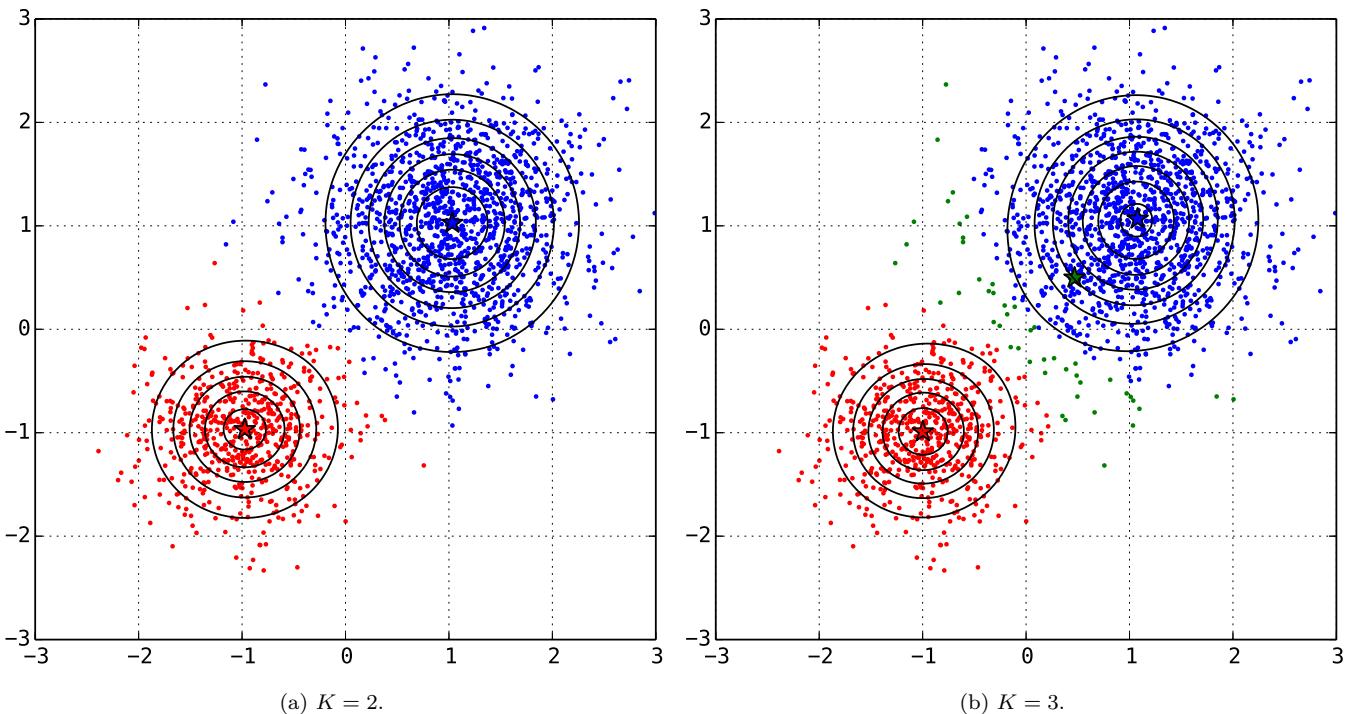


Fig. 6.4: variational Bayesian inference on a Gaussian mixture model.

正則化の恩恵により、集団の個数を過剰に設定した場合でも、余剰の集団の加重が徐々に低下し、過学習が抑制される。

6.6 母数の事後分布の導出

変分ベイズ法の式 (6.26) に対し、第 6.5 節で設定した共役事前分布を代入する。まず、全ての変数の結合確率を求める。

$$p(\mathbf{x}, z, w, \boldsymbol{\mu}, S) = p(\mathbf{x}, z | w, \boldsymbol{\mu}, S) p(w, \boldsymbol{\mu}, S) = p(\mathbf{x} | z, \boldsymbol{\mu}, S) p(z | w) p(w) p(\boldsymbol{\mu} | S) p(S). \quad (6.34)$$

E-step を導く。母数 θ の事前分布を固定し、潜在変数 z の分布を最適化する操作なので、その間に式 (6.35) が成立する。

$$f(z) \propto \exp \mathbb{E}_g[\log p(\mathbf{x}, z | w, \boldsymbol{\mu}, S)] = \exp \sum_{n=1}^N \sum_{k=1}^K z_{nk} \left\{ \mathbb{E}_w[\log w_k] + \mathbb{E}_{\boldsymbol{\mu}, S}[\log \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k)] \right\} \propto \prod_{n=1}^N \prod_{k=1}^K \gamma_{nk}^{z_{nk}}. \quad (6.35)$$

式 (6.35) に現れる、加重 w の対数の期待値は、式 (6.36) となる。関数 ψ は**ディガンマ関数**で、関数 Γ の対数微分である。

$$\mathbb{E}_w[\log w_k] = \frac{1}{B(\alpha)} \frac{\partial}{\partial \alpha_k} \int \prod_{j=1}^K w_j^{\alpha_j-1} dw = \frac{\partial}{\partial \alpha_k} \log B(\alpha) = \psi(\alpha_k) - \psi\left(\sum_{j=1}^K \alpha_j\right). \quad (6.36)$$

式 (6.35) に現れる、正規分布の対数の期待値は、式 (6.2) の正規分布の確率密度関数より、式 (6.37) の形に分解できる。

$$\mathbb{E}_{\boldsymbol{\mu}, S}[\log \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, S_k)] = -\frac{1}{2} \mathbb{E}_{\boldsymbol{\mu}, S}[D \log 2\pi + \log |S_k| + {}^t(\mathbf{x}_n - \boldsymbol{\mu}_k) S_k^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_k)]. \quad (6.37)$$

式 (6.37) に現れる、行列式 $|S|$ の対数の期待値は、式 (6.32) の確率密度関数を母数 ν_k で偏微分すれば、式 (6.38) となる。

$$\mathbb{E}_S[\log |S_k|] = 2 \int \frac{\partial \tilde{\mathcal{W}}}{\partial \nu_k} \frac{\mathcal{W}}{\tilde{\mathcal{W}}} dS - 2 \int \frac{\partial \mathcal{W}}{\partial \nu_k} dS = \frac{2}{\tilde{\mathcal{W}}} \frac{\partial \tilde{\mathcal{W}}}{\partial \nu_k} = -D \log 2 - \log |W_k| - \sum_{d=0}^{D-1} \psi\left(\frac{\nu_k - d}{2}\right). \quad (6.38)$$

式 (6.37) に現れる、行列積の期待値は、母数 $\boldsymbol{\mu}$ が、式 (6.31) の正規分布に従う事実と因数分解により、式 (6.39) となる。

$$\mathbb{E}_{\boldsymbol{\mu}, S}[{}^t(\mathbf{x}_n - \boldsymbol{\mu}_k) S_k^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_k)] = \nu_k {}^t(\mathbf{x}_n - \mathbf{m}_k) W_k (\mathbf{x}_n - \mathbf{m}_k) + \frac{D}{\sigma_k}. \quad (6.39)$$

M-step を導く。事後確率 γ を式 (6.40) に代入し、事前分布と事後分布の共役性に注意して、母数の事後分布を求めよう。

$$g(\theta) \propto \exp \left\{ \mathbb{E}_z[\log p(\mathbf{x}, z | w, \boldsymbol{\mu}, S)] + \log p(w) + \log p(\boldsymbol{\mu} | S) + \log p(S) \right\}. \quad (6.40)$$

式 (6.40) で、変数 x, z の結合確率の対数の期待値は、式 (6.2) の正規分布と式 (6.29) の多項分布より、式 (6.41) となる。

$$\mathbb{E}_z[\log p(\mathbf{x}, z | w, \boldsymbol{\mu}, S)] = -\frac{1}{2} \sum_{k=1}^K \sum_{n=1}^N \gamma_{nk} \left\{ D \log 2\pi + |S_k| + {}^t(\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k) S_k^{-1} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k) - 2 \log w_k \right\}. \quad (6.41)$$

共役性より、母数 θ の事後分布は、事前分布の母数 ϕ を、推定値 $\hat{\phi}$ に置換した場合と等値であり、式 (6.42) が成立する。

$$\mathbb{E}_z[\log p(\theta | \mathbf{x}, z)] = \log p(\theta | \hat{\phi}) = \mathbb{E}_z[\log p(\mathbf{x}, z | \theta)] + \log p(\theta | \phi) - \mathbb{E}_z[\log p(\mathbf{x}, z)]. \quad (6.42)$$

式 (6.42) より、母数 α, σ, ν の推定値に対して、式 (6.43) が成立する。変数 N_k は、集団 C_k の要素の個数の期待値を表す。

$$\hat{\alpha}_k - \alpha_k = \hat{\sigma}_k - \sigma_k = \hat{\nu}_k - \nu_k = N_k = \sum_{n=1}^N z_{nk}. \quad (6.43)$$

母数 \mathbf{m} の場合は、式 (6.44) が成立する。期待値最大化法の式 (6.11) を考えれば、事前分布と標本平均の加重平均である。

$$\hat{\mathbf{m}}_k = \frac{1}{\hat{\sigma}_k} \left(\sigma_k \mathbf{m}_k + \sum_{n=1}^N \gamma_{nk} \mathbf{x}_n \right). \quad (6.44)$$

母数 W の場合は、式 (6.45) が成立する。これも、式 (6.45) の右辺に着目すれば、事前分布と標本分散の加重平均である。

$$\hat{W}_k^{-1} = W_k^{-1} + \sum_{n=1}^N \gamma_{nk} {}^t \mathbf{x}_n \mathbf{x}_n + \sigma_k {}^t \mathbf{m}_k \mathbf{m}_k - \hat{\sigma}_k \hat{\mathbf{m}}_k {}^t \hat{\mathbf{m}}_k. \quad (6.45)$$

式 (6.45) の導出では、分散 W の**コレスキーフィルタ**により下三角行列 T が存在して、式 (6.46) が成立する性質を利用した。

$${}^t \mathbf{x} W \mathbf{x} = ({}^t \mathbf{x} T) ({}^t T \mathbf{x}) = {}^t ({}^t T \mathbf{x}) ({}^t T \mathbf{x}) = \text{tr} (({}^t T \mathbf{x}) ({}^t T \mathbf{x})) = \text{tr} ({}^t T ({}^t \mathbf{x} T)) = \text{tr} (({}^t \mathbf{x} T) W) = \text{tr} (({}^t \mathbf{x}) W). \quad (6.46)$$

事前分布と最尤推定の最適値の加重平均が現れる点には、式 (6.27) で議論した、事前分布による正則化の効果が窺える。

6.7 変分ベイズ推定の実装

第6.5節の議論に基づき、変分ベイズ推定を実装する。まず、本体を実装する。引数は、期待値最大化法と同等である。式(6.43)より、母数 α, σ, ν は、初期値を揃えると、以後の最適化を通じて常に同じ値になる。そこで、同じ変数にした。

```
class VB(val x: Seq[Seq[Double]], val mm: GMM, epochs: Int = 1000, W: Double = 1) {
    val n = Array.fill(mm.k)(1.0 / mm.k)
    val w -> m = (Array.fill(mm.k, mm.d)(W), Array.fill(mm.k, mm.d)(math.random))
    for(epoch <- 1 to epochs) new MstepGMM(this, mm, new EstepGMM(this, mm).post)
}
```

*E-step*を実装する。式(6.35)に基づき、潜在変数 z の事後確率 γ を計算する。最後に、総和で事後確率 γ を規格化する。

```
class EstepGMM(vb: VB, mm: GMM) {
    val eq35 = vb.n.map(Digamma).map(_-Digamma(vb.n.sum))
    val eq3A = vb.n.map(n=>0.to(mm.d-1).map(d=>(n-d)/2).map(Digamma))
    val eq36 = eq3A.zip(vb.w).map(_.sum_-.map(math.log).sum).map(_/2)
    def wish = vb.x.toArray.map(_.toArray).map(vb.m-_).map(d=>d.mul(d).div(vb.w))
    def eq34 = wish.map(_.zip(vb.n).map(_-.sum/2*_))+eq35+eq36-vb.n.map(mm.d/_/2)
    def post = eq34.map(_.map(math.exp)).map(x=>x.map(_/x.sum)).toSeq.transpose
}
```

*M-step*を実装する。期待値最大化法の実装を流用して、母数の最尤推定値を計算し、事前分布との加重平均を計算する。

```
class MstepGMM(vb: VB, mm: GMM, post: Seq[Seq[Double]]) {
    new EM(vb.x, mm, 0).mstep(post)
    val eq11 = post.map(_.sum).toArray
    val eq38 = vb.n.zip(eq11).map(_+_)
    val eq39 = vb.m.mul(vb.n).div(eq38).add(mm.m.mul(eq11).div(eq38))
    val eq41 = vb.m.mul(vb.m).mul(vb.n).sub(eq39.mul(eq39).mul(eq38))
    val eq40 = mm.s.add(mm.m.mul(mm.m)).mul(eq11).add(vb.w.add(eq41))
    eq38.copyToArray(vb.n)
    eq39.zip(vb.m).foreach(_.copyToArray(_))
    eq40.zip(vb.w).foreach(_.copyToArray(_))
}
```

*M-step*では、平均や分散の配列の四則演算が頻繁に現れる。簡潔な実装を目指し、暗黙の型変換で四則演算を実現した。

```
implicit class Vector(x: Array[Array[Double]]) {
    def +(y: Array[Double]) = x.map(_.zip(y).map(_+_))
    def -(y: Array[Double]) = x.map(_.zip(y).map(_-_))
    def add(y: Array[Double]) = x.zip(y).map((x,y) => x.map(_+y))
    def sub(y: Array[Double]) = x.zip(y).map((x,y) => x.map(_-y))
    def mul(y: Array[Double]) = x.zip(y).map((x,y) => x.map(_*y))
    def div(y: Array[Double]) = x.zip(y).map((x,y) => x.map(_/y))
    def add(y: Array[Array[Double]]) = x.zip(y).map(_.zip(_).map(_+_));
    def sub(y: Array[Array[Double]]) = x.zip(y).map(_.zip(_).map(_-_))
    def mul(y: Array[Array[Double]]) = x.zip(y).map(_.zip(_).map(_*_))
    def div(y: Array[Array[Double]]) = x.zip(y).map(_.zip(_).map(_/_))
}
```

最後に、ディガンマ関数を実装する。詳細は省くが、**ワイエルシュトラスの無限乗積表示**を利用して、簡単に計算できる。

```
object Digamma extends Function[Double, Double] {
    def apply(x: Double): Double = {
        var index -> value = (x, 0.0)
        def d = 1.0 / (index * index)
        while(index < 49) (value -= 1 / index, index += 1)
        val s = d * (1.0 / 12 - d * (1.0 / 120 - d / 252))
        (value + math.log(index) - 0.5 / index - s)
    }
}
```