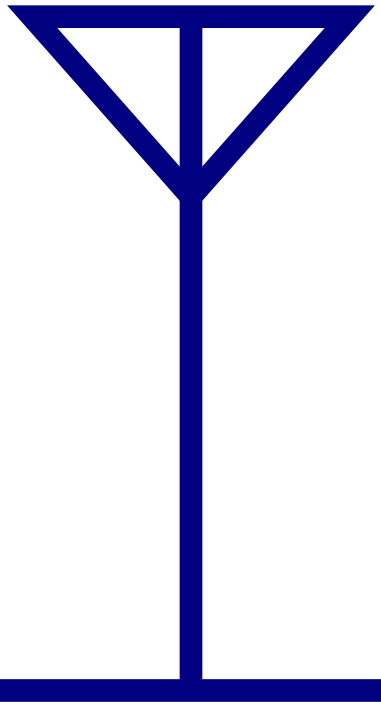
Journal of Hamradio Informatics No.3

Scalaで実装するパターン認識と機械学習

Scala's Pattern Recognition & Machine Learning



無線部開発班　**平成 29 年 11 月 26 日改訂**

http://pafelog.net

目次

第1章　回帰とクラス分類の基礎

第 1 章では、機械学習の初歩的な事例として**線型回帰**と**最近傍法**を実装し、機械学習の感覚的理解を目指す。機械学習とは、$\boldsymbol{y} = f(\boldsymbol{x})$ に従うデータ集合 $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}$ に対し、関数 f を推定するアルゴリズムの総称である。なお、訓練データに $\{\boldsymbol{y}_n\}$ が明示的に含まれる場合を**教師あり学習**と呼び、それ以外を**教師なし学習**と呼ぶ。

1.1　線型回帰

教師あり学習の中でも、出力すべき変数 $\boldsymbol{y}$ が連続値を取る場合を**回帰**と呼び、線型回帰はその初歩と言える。線型回帰は、式 (1.1) に示す通り、従属変数 y を自由変数 x_k に加重 w_k を掛けた和で説明するモデルである。

$$y = f(\boldsymbol{x}) = w_0 + \sum_{k=1}^{K} w_k x_k = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_K \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_K \end{pmatrix} = {}^t\boldsymbol{w}\ \boldsymbol{x}. \tag{1.1}$$

また、何らかの非線型な写像 $\phi_k(\boldsymbol{x})$ の線型結合で回帰を行う式 (1.2) のモデルを**線型基底関数モデル**と呼ぶ。

$$y = f(\boldsymbol{x}) = \sum_{k=0}^{K} w_k \phi_k(\boldsymbol{x}) = {}^t\boldsymbol{w}\ \boldsymbol{\phi}(\boldsymbol{x}). \tag{1.2}$$

Fig. 1.1 は $y \sim \mathcal{N}\left(y|x^3 - 3x, 2500\right)$ に従う $\{x_n, y_n\}$ に $\phi_k(x) = x^k$ の線型基底関数モデルを適用した例である。

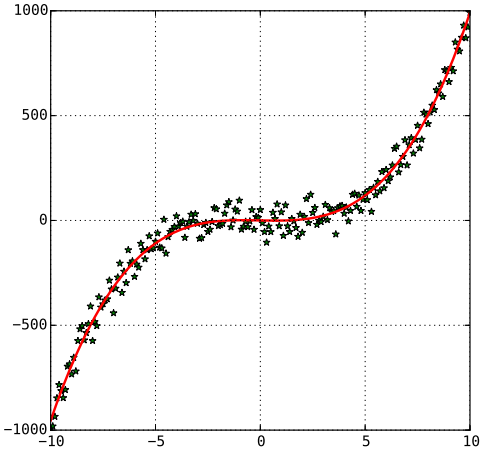


Fig. 1.1: linear basis function model.

線型基底関数モデルでは、他にも $\phi_k(x) = \mathcal{N}\left(x|\mu_k, \sigma_k^2\right)$ を採用すれば、μ_k 近傍の局所的な曲線を表現できる。

$$y = f(\boldsymbol{x}) = \begin{cases} \displaystyle\sum_{k=0}^{K} w_k x^k & \text{where } \phi_k(x) = x^k, \\ \displaystyle\sum_{k=0}^{K} \frac{w_k}{\sqrt{2\pi\sigma_k^2}} \exp\left\{-\frac{(x-\mu_k)^2}{2\sigma_k^2}\right\} & \text{where } \phi_k(x) = \mathcal{N}\left(x|\mu_k, \sigma_k^2\right). \end{cases} \tag{1.3}$$

実際の観測にはノイズ ε_n が重畳する。ノイズは釣鐘型に分布するので、正規分布を仮定して式 (1.4) を得る。

$$y = f(\boldsymbol{x}) + \varepsilon \sim p(y|\boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}\left(y|f(\boldsymbol{x}), \beta^{-1}\right). \tag{1.4}$$

なお、β は分散の逆数で、ノイズの**精度**である。観測された標本 $\{\boldsymbol{x}_n, y_n\}$ の出現確率は、式 (1.5) で求まる。

$$p(\{y_n\}\,|\,\{\boldsymbol{x}_n\}, \boldsymbol{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}\left(y_n|{}^t\boldsymbol{w}\ \boldsymbol{\phi}(\boldsymbol{x}_n), \beta^{-1}\right). \tag{1.5}$$

式 (1.5) を**尤度**と呼ぶ。尤度は、与えられた訓練データを良く表現する**母数**としての加重 $\boldsymbol{w}$ の妥当性を示す。線型回帰の学習は、尤度の最大化を目指す。今回は、計算の容易性から、式 (1.6) の対数尤度を最大化する。

$$\log p(\{y_n\}\,|\,\{\boldsymbol{x}_n\}, \boldsymbol{w}, \beta) = \frac{N}{2}\log\beta - \frac{N}{2}\log 2\pi - \frac{\beta}{2}\sum_{n=1}^{N}\{y_n - {}^t\boldsymbol{w}\ \boldsymbol{\phi}(\boldsymbol{x}_n)\}^2. \tag{1.6}$$

式 (1.6) から定数項を取り除き、加重 $\boldsymbol{w}$ に依存する項を取り出すと、式 (1.7) の 2 乗誤差関数 $E(\boldsymbol{w})$ を得る。式 (1.7) の最小化は、**最小二乗法**とも呼ばれる。さて、誤差 $E(\boldsymbol{w})$ を最小化する具体的な方法を検討しよう。

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{y_n - \sum_{k=0}^{K} w_k\phi_k(\boldsymbol{x}_n)\right\}^2. \tag{1.7}$$

誤差 $E(\boldsymbol{w})$ は**連続関数**なので、勾配 $\nabla E(\boldsymbol{w})$ が 0 の点で最小になる。ここで、式 (1.8) の**勾配法**を導入する。勾配法とは、加重 $\boldsymbol{w}$ を極値に向けて動かす操作を繰り返し、最終的に $\nabla E(\boldsymbol{w}) = 0$ で収束させる手法である。

$$\boldsymbol{w}' = \boldsymbol{w} - \eta\nabla E(\boldsymbol{w}) = \boldsymbol{w} + \eta\sum_{n=1}^{N}\{y_n - {}^t\boldsymbol{w}\ \boldsymbol{\phi}(\boldsymbol{x}_n)\}\boldsymbol{\phi}(\boldsymbol{x}_n). \tag{1.8}$$

係数 η は**学習率**と呼ばれ、加重 $\boldsymbol{w}$ の発散を防止するため、$\eta\nabla E(\boldsymbol{w}) \ll \boldsymbol{w}$ となるように調整する必要がある。勾配法による線型回帰を `Regression` クラスに実装する。引数 `e` は学習率 η で、`XY` は標本 $\{x_n, y_n\}$ を表す。

LinearRegression.scala

```
class Regression(e: Double, XY: Seq[(Double, Double)], p: Seq[Double=>Double]) {
  val w = Array.fill[Double](p.size)(0)
```

配列 `p` は、写像 $\phi_k(x)$ を格納する長さ K の列 $\{\phi_k\}$ である。下記の `apply` メソッドは、式 (1.2) を計算する。

LinearRegression.scala

```
  def apply(x: Double) = w.zip(p.map(_(x))).map{case (w,x)=>w*x}.sum
```

最後に、式 (1.8) による加重 $\boldsymbol{w}$ の更新を実装する。反復回数を固定せずに、収束判定を行うと実用的である。

LinearRegression.scala

```
  for(n<-1 to 1000; (x,y) <- XY; k<-0 until p.size) w(k) += e * (y-this(x)) * p(k)(x)
}
```

完成した `Regression` クラスに $y = x^3 + \varepsilon$ の標本を与えて、基底 $\{x^3, x^2, x, 1\}$ で線型回帰を行う例を示す。

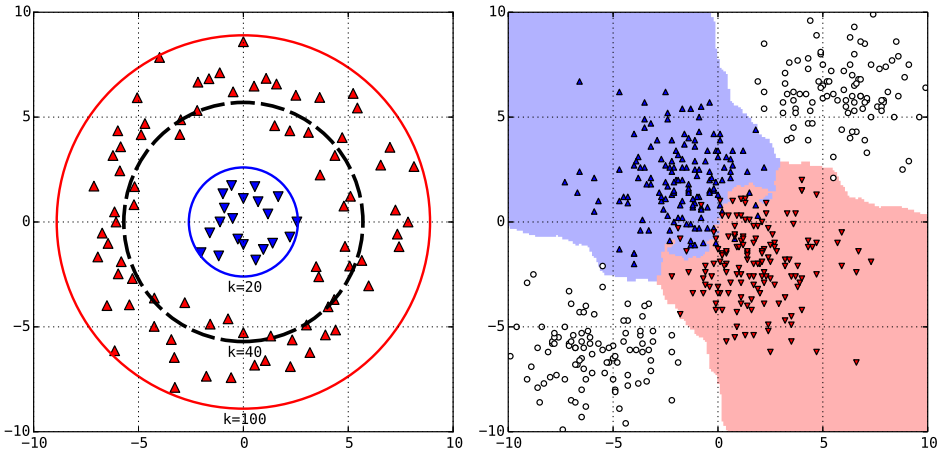
LinearRegression.scala

```
val pts = -10.to(10).map(x=>(x.toDouble, math.pow(x,3) + util.Random.nextGaussian))
val reg = new Regression(0.000001, pts, 0.to(3).map(k=>(x: Double)=>math.pow(x,k)))
```

第 5 章で学ぶ**ニューラルネットワーク**は、活性化関数を追加した線型基底関数モデルと本質的に等価である。

1.2 最近傍法

教師あり学習の中でも、従属変数 $\boldsymbol{y}$ が離散値を取る場合を**クラス分類**と呼び、最近傍法はその初歩と言える。Fig. 1.2(a) に示す最近傍法は、未知の点を分類する際に近傍の k 点を参考にする。k の決め方が重要である。



(a) k nearest neighbor diameters. (b) $k=10$ region segmentation.

Fig. 1.2: k nearest neighbor model.

最近傍法は**遅延学習**とも呼ばれ、事前の学習処理が不要で、分類時に初めて標本を参照する点が特徴である。下記の KNN クラスは、未分類の点と訓練データの各点との距離を計算し、最近傍の k 個の点で多数決を行う。

NearestNeighbor.scala

```
class KNN[D,T](k: Int, data: Seq[(D,T)], d: (D,D)=>Double) {
```

引数 d には距離関数 d を指定する。例えば、**平方ユークリッド距離**を使うには、下記の quad 関数を与える。

NearestNeighbor.scala

```
def quad(a: Seq[Double], b: Seq[Double]) = (a, b).zipped.map(_-_).map(x => x * x).sum
```

距離関数 d の決め方は重要な課題である。典型的には平方ユークリッド距離や**マンハッタン距離**を採用する。厳密には、式 (1.9) の**距離の公理**を満たす必要があるが、距離の比較ができれば、厳密な定義は不要である。

$$\begin{cases} d(\boldsymbol{x},\boldsymbol{y}) \geq 0, \\ \boldsymbol{x}=\boldsymbol{y} \Leftrightarrow d(\boldsymbol{x},\boldsymbol{y})=0, \\ d(\boldsymbol{x},\boldsymbol{y}) = d(\boldsymbol{y},\boldsymbol{x}), \\ d(\boldsymbol{x},\boldsymbol{y}) \leq d(\boldsymbol{x},\boldsymbol{z}) + d(\boldsymbol{z},\boldsymbol{y}). \end{cases} \tag{1.9}$$

下記の apply メソッドは、座標 $\boldsymbol{x}$ 近傍の k 点を引数 data で与えた標本 $\{\boldsymbol{x}_n, y_n\}$ から探し、多数決を採る。

NearestNeighbor.scala

```
  def apply(x: D) = data.sortBy(s=>d(x,s._1)).take(k).groupBy(_._2).maxBy(_._2.size)._1
}
```

Fig. 1.2(b) は、混合正規分布に従う標本を KNN クラスで学習して、変数 $\boldsymbol{x}$ の空間を塗り分けた結果である。実用的には、距離計算の計算量を抑えるため、R 木などの**空間データベース**で探索空間を局限すべきである。

第2章　決定木の学習と汎化性能

式 (2.1) は、気象条件 $\boldsymbol{x}$ に対して質問と条件分岐を繰り返し、海水浴の是非 y を判断する**決定木**の例である。

$$y \approx f(\boldsymbol{x}) = \begin{cases} 0 & \text{if wavy}(\boldsymbol{x}) = 1 \\ \text{otherwise} \begin{cases} 0 & \text{if rain}(\boldsymbol{x}) = 1 \\ 1 & \text{if rain}(\boldsymbol{x}) = 0 \end{cases} & \text{if wavy}(\boldsymbol{x}) = 0 \end{cases} \tag{2.1}$$

理想的な決定木は、簡潔明瞭である。故に、決定木の学習は標本 $\{\boldsymbol{x}_n\}$ に対する質問回数の最小化を目指す。質問回数の最小化は、第 2.1 に述べる**情報量の加法性**を勘案すれば、質問の**情報利得**の最大化と同義である。

2.1　情報利得の最大化

ある確率分布に従う記号列 $\{y_n\}$ を出力する装置を**情報源**と呼び、特定の情報 y の価値 $I(y)$ を**情報量**と呼ぶ。真に価値ある情報は、稀有な筈である。試みに、情報量 $I(y)$ を記号 y の出現確率 $P(y)$ の反比例で定義する。

$$I(y) = \frac{1}{P(y)}. \tag{2.2}$$

式 (2.2) の定義は完璧に思えるが、記号列 $\{y_n\}$ の情報量 $I(\{y_n\})$ を計算すると、式 (2.3) の違和感に気付く。

$$I(\{y_n\}) = \prod_{n=1}^{N} \frac{1}{P(y_n)} = \prod_{n=1}^{N} I(y_n). \tag{2.3}$$

情報量が情報の価値を表す量であるなら、情報量 $I(\{y_n\})$ は、情報量 $\{I(y_n)\}$ の和になって然るべきである。この直観を情報量の加法性と呼ぶ。そこで、対数関数を利用して、情報量 $I(y)$ の定義を式 (2.4) で修正する。

$$I(y) = \log_2 \frac{1}{P(y)} = -\log_2 P(y) \geq 0. \tag{2.4}$$

また、情報源 Y に対し、情報量 $I(y)$ の期待値 $H(Y)$ を定義する。期待値 $H(Y)$ は**エントロピー**と呼ばれる。

$$H(Y) = \sum_{y \in Y} P(y) I(y) = -\sum_{y \in Y} P(y) \log_2 P(y) \geq 0. \tag{2.5}$$

なお、エントロピー $H(Y)$ が 0 になる状況は、情報源 Y が常に同じ記号 y のみを出力する場合に限られる。記号 y の集合 Y を質問 Q で分割し、K 個の部分集合 $\{Y_k\}$ を得た場合、式 (2.6) の $G(Q)$ を情報利得と呼ぶ。

$$G(Q) = H(Y) - H(Y|Q) = H(Y) - \sum_{k=1}^{K} \frac{|Y_k|}{|Y|} H(Y_k) \geq 0. \tag{2.6}$$

集合 Y_k は、決定木の**子ノード**に該当する。Y_k は更なる質問 Q_k で分割されて、再帰的に木構造を構築する。決定木を辿り、質問を重ねる度に、エントロピーは平均的に減少し、0 に収束した時点で y の値が確定する。

DecisionTree.scala

```
trait Node[T] {
  def apply(x: Seq[Int]): T
}
```

決定木の動作は煩雑なので、1 回の質問を複数の Node の実装クラスに分解することで、実装を簡素化する。Question クラスは、引数 Y に説明変数と従属変数の標本 $\{\boldsymbol{x}_n, y_n\}$ を受け取り、決定木を再帰的に構築する。

DecisionTree.scala

```
case class Question[T](Y: Seq[(Seq[Int], T)]) extends Node[T] {
  lazy val freqs = Y.groupBy(_._2).map(_._2.size.toDouble / Y.size)
  lazy val ent = freqs.map(f => -f * math.log(f)).sum / math.log(2)
  lazy val major = Y.groupBy(_._2).maxBy(_._2.size)._1
  lazy val v = Y.head._1.indices.map(Variable(Y, _)).minBy(_.t.ent)
  def apply(x: Seq[Int]) = if(ent - v.t.ent < 1e-5) major else v(x)
}
```

freqs は $P(y)$ の値を記憶し、ent は $H(Y)$ の値を記憶する。major は $P(y)$ を最大化する y の値を記憶する。apply メソッドは分類を行う。学習した質問に照らして、変数 $\boldsymbol{x}$ に対応する変数 y の値を再帰的に推論する。

DecisionTree.scala

```
case class Variable[T](Y: Seq[(Seq[Int], T)], axis: Int) extends Node[T] {
  val t = Y.map(_._1(axis)).distinct.map(Division(Y, axis, _)).minBy(_.ent)
  def apply(x: Seq[Int]) = t(x)
}
```

Variable クラスの引数 axis は分岐する軸を表し、その閾値は Division クラスの引数 value で指示する。引数 axis や引数 value には、式 (2.6) の条件付きエントロピー $H(Y|Q)$ を最小化する軸と閾値が選ばれる。

DecisionTree.scala

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

ent は $H(Y|Q)$ を保持する。apply メソッドは、条件分岐を行って、子ノードの apply メソッドを実行する。Fig. 2.1 は、混合正規分布に従う標本を Question クラスで学習し、変数 $\boldsymbol{x}$ の空間を塗り分けた結果である。

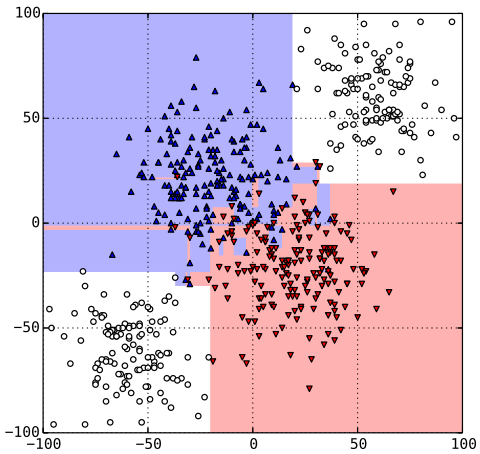


Fig. 2.1: region segmentation by a decision tree.

学習結果は過剰に複雑な境界線を描き、標本には忠実だが、母集団から乖離している。これを**過学習**と呼ぶ。過学習を防ぐには、決定木の枝刈りを行うか、第 2.2 で学ぶアンサンブル学習により**汎化性能**の改善を図る。

2.2 汎化誤差の最小化

標本 $\{\boldsymbol{x}_n, y_n\}$ から得た関数 $\hat{f}(\boldsymbol{x})$ と真の $f(\boldsymbol{x})$ の間には**汎化誤差**が生じ、期待値を分解すると式 (2.7) を得る。

$$\int_{\boldsymbol{x}} P(\boldsymbol{x})(y-\hat{f}(\boldsymbol{x}))^2 d\boldsymbol{x} = \mathbf{V}[\,y-f(\boldsymbol{x})\,] + \left(\mathbf{E}\left[\,\hat{f}(\boldsymbol{x})\,\right] - f(\boldsymbol{x})\right)^2 + \mathbf{V}\left[\,\hat{f}(\boldsymbol{x})\,\right]. \tag{2.7}$$

T 個の分類器 $\{\hat{f}_t(\boldsymbol{x})\}$ を**弱学習器**と称して糾合し、投票で誤差の抑制を図る手法を**アンサンブル学習**と呼ぶ。

$$\hat{f}(\boldsymbol{x}) = \arg\max_k \frac{1}{T}\sum_{t=1}^{T} \mathbb{I}\left(\hat{f}_t(\boldsymbol{x}) = k\right) \text{ where } \mathbb{I}\left(\hat{f}_t(\boldsymbol{x}) = k\right) = \begin{cases} 1 & \text{if } \hat{f}_t(\boldsymbol{x}) = k, \\ 0 & \text{if } \hat{f}_t(\boldsymbol{x}) \neq k. \end{cases} \tag{2.8}$$

中でも**バギング**と呼ばれる手法では、式 (2.9) に示す相関を抑制することで、式 (2.7) の第 3 項の抑制を図る。

$$\mathbf{V}\left[\,\hat{f}(\boldsymbol{x})\,\right] = \frac{1}{T^2}\sum_{i=1}^{T}\sum_{j=1}^{T} \mathbf{C}[\,\mathbb{I}\,(f_i(\boldsymbol{x}) = k)\,, \mathbb{I}\,(f_j(\boldsymbol{x}) = k)\,]\,. \tag{2.9}$$

下記の `Bagging` クラスは、要素の重複を許して濃度 N の標本を T 通り抽出し、式 (2.9) の相関を抑制する。

`DecisionTree.scala`

```
case class Bagging[T](Y: Seq[(Seq[Int], T)], T: Int, N: Int) extends Node[T] {
  val t = Seq.fill(T)(Question(Seq.fill(N)(Y(util.Random.nextInt(Y.size)))))
  def apply(x: Seq[Int]) = t.map(_(x)).groupBy(identity).maxBy(_._2.size)._1
}
```

他方、**ブースティング**と呼ばれる手法では、弱学習器 $\hat{f}_t(\boldsymbol{x})$ は $\hat{f}_{t-1}(\boldsymbol{x})$ が判断を誤る点を重点的に学習する。$\hat{f}_t(\boldsymbol{x})$ と $\hat{f}_{t-1}(\boldsymbol{x})$ は対等でなく、その信頼度に基づく加重 $\{w_t\}$ を付与され、式 (2.10) に示す加重投票を行う。

$$\hat{f}(\boldsymbol{x}) = \arg\max_k \sum_{t=1}^{T} w_t \mathbb{I}\left(\hat{f}_t(\boldsymbol{x}_n) = k\right). \tag{2.10}$$

学習の目標は、式 (2.11) に示す**指数誤差**の期待値を最小化する弱学習器 $\{\hat{f}_t(\boldsymbol{x})\}$ と加重 $\{w_t\}$ の設定にある。

$$\mathbf{E}\left[\sum_{n=1}^{N} \exp\left\{-\frac{1}{K}\sum_{k=1}^{K} \mathbb{I}_k(y_n)\,\mathbb{I}_k\left(\hat{f}(\boldsymbol{x})\right)\right\}\right] \text{ where } \mathbb{I}_k(y) = \begin{cases} 1 & \text{if } y = k, \\ \dfrac{1}{1-K} & \text{if } y \neq k. \end{cases} \tag{2.11}$$

最後の弱学習器 $\hat{f}_T(\boldsymbol{x})$ の学習に着目し、標本の分布 $P_T(\boldsymbol{x}, y)$ を導入して、式 (2.11) を式 (2.12) に変形する。

$$\mathbf{E}\left[\sum_{n=1}^{N} P_T(\boldsymbol{x}_n, y_n) \exp\left\{-\frac{1}{K}\sum_{k=1}^{K} \mathbb{I}_k(y_n)\, w_T \mathbb{I}_k\left(\hat{f}_T(\boldsymbol{x}_n)\right)\right\}\right]. \tag{2.12}$$

式 (2.11) の変形の過程で現れる分布 $P_T(\boldsymbol{x}, y)$ は、弱学習器 $\hat{f}_{T-1}(\boldsymbol{x})$ が判断を誤る点を重点的に学習させる。

$$P_T(\boldsymbol{x}_n, y_n) = \exp\left\{-\frac{1}{K}\sum_{k=1}^{K} \mathbb{I}_k(y_n) \sum_{t=1}^{T-1} w_t \mathbb{I}_k\left(\hat{f}_t(\boldsymbol{x}_n)\right)\right\}. \tag{2.13}$$

式 (2.13) に従う標本を抽出する操作は、**ノイマンの棄却法**を利用して、下記の `Resample` クラスに実装する。

`DecisionTree.scala`

```
case class Resample[T](Y: Seq[(Seq[Int], T)], P: Seq[Double]) extends Node[T] {
  def reject(i: Int) = if(util.Random.nextDouble * P.max < P(i)) Y(i) else null
  val data = new collection.mutable.ArrayBuffer[(Seq[Int], T)]
  while(data.size < P.size) data += reject(util.Random.nextInt(P.size)) -= null
  val quest = Question(data)
  val error = Y.zip(P).map{case ((x, y), p) => if(quest(x) != y) p else 0}.sum
  def apply(x: Seq[Int]) = quest(x)
}
```

下記の AdaStage クラスは M 個の候補 $\{\hat{f}_{tm}(\boldsymbol{x})\}$ から誤差 E_t が最小の候補を選び、弱学習器 $\hat{f}_t(\boldsymbol{x})$ とする。

DecisionTree.scala

```
case class AdaStage[T](Y: Seq[(Seq[Int], T)], P: Seq[Double], M: Int) extends Node[T] {
  val best = List.fill(M)(Resample(Y, P.map(_ / P.sum))).minBy(_.error)
  val W = math.log((1 / best.error - 1) * (Y.map(_._2).toSet.size - 1))
  def isOK = best.error < 0.5
  def apply(x: Seq[Int]) = best(x)
  def apply(x: Seq[Int], y: T): Double = if(best(x) == y) W else 0
  val next = Y.zip(P).map{case ((x, y), p) => p * math.exp(W - this(x, y))}
}
```

W は式 (2.12) を最小化する加重 W_T であり、式 (2.14) で計算する。ただし、K はクラスの異なり数である。

$$\hat{w}_T = \left\{\log\left(\frac{1}{E_T} - 1\right) + \log(K-1)\right\} \text{ where } E_T = \sum_{n=1}^{N} P(\boldsymbol{x}_n, y_n)\mathbb{I}\left(\hat{f}_T(\boldsymbol{x}_n) \neq y_n\right). \tag{2.14}$$

下記の AdaBoost クラスは、弱学習器 $\hat{f}_t(\boldsymbol{x})$ の誤差 E_t が 0.5 を上回るまで $\hat{f}_t(\boldsymbol{x})$ を生成し、加重投票を行う。

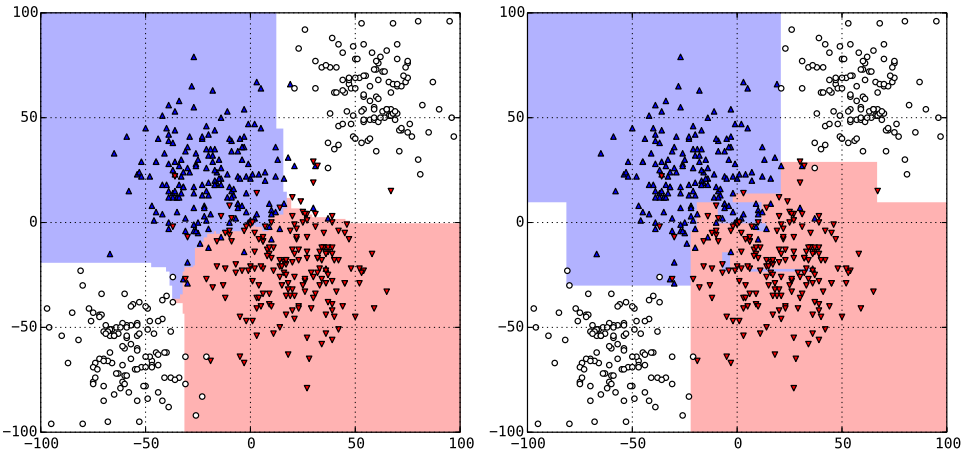
DecisionTree.scala

```
case class AdaBoost[T](Y: Seq[(Seq[Int], T)], M: Int) extends Node[T] {
  val stages = Seq(AdaStage(Y, Y.map(_ => 1.0 / Y.size), M)).toBuffer
  while(stages.last.isOK) stages += AdaStage(Y, stages.last.next, M)
  def apply(x: Seq[Int], y: T): Double = stages.init.map(_(x, y)).sum
  def apply(x: Seq[Int]) = Y.map(_._2).distinct.maxBy(this(x, _))
}
```

式 (2.11) を最小化する $\hat{f}(\boldsymbol{x})$ は式 (2.15) を満たす。故に、指数誤差の最小化は式 (2.7) の第 2 項を抑制する。

$$\hat{f}(\boldsymbol{x}) = \arg\min_k (K-1)\left\{\log P(y=k|\boldsymbol{x}) - \frac{1}{K}\sum_{k=1}^{K}\log P(y=k|\boldsymbol{x})\right\} = \arg\min_k P(y=k|\boldsymbol{x}). \tag{2.15}$$

比較のため、Fig. 2.1 と同じ標本を Bagging クラスと AdaBoost クラスに学習させた結果を Fig. 2.2 に示す。



(a) Bagging class. (b) AdaBoost class.

Fig. 2.2: region segmentation by ensemble learning.

バイアスバリアンス理論では、式 (2.7) の汎化誤差の第 2 項を**バイアス**と呼び、第 3 項を**バリアンス**と呼ぶ。決定木は、バイアスが低くバリアンスが高いモデルなので、ブースティングよりもバギングが効果的である。

第3章　混合正規分布と最尤推定

第 3 章では、観測データ $\boldsymbol{x}$ が従う何らかの確率分布 $P(\boldsymbol{x})$ を仮定し、分布 $P(\boldsymbol{x})$ の母数を求める手順を学ぶ。単峰型の分布としては正規分布が代表的だが、Fig. 3.1 に示す**混合正規分布**なら多峰型の分布を表現できる。

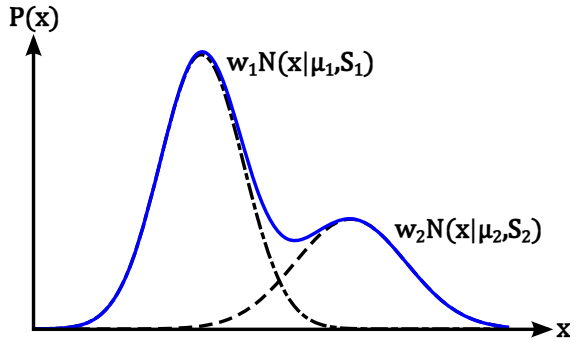


Fig. 3.1: Gaussian mixture model.

混合正規分布は K 個の正規分布の線型和である。変数 $\boldsymbol{x} \in \mathbb{R}^D$ は、確率 w_k で正規分布 $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, S_k)$ に従う。

$$P(\boldsymbol{x}) = \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, S_k) \text{ where } \sum_{k=1}^{K} w_k = 1. \tag{3.1}$$

$\boldsymbol{\mu}_k$ と S_k は k 番目の正規分布の平均と**分散共分散行列**である。正規分布 $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, S)$ は式 (3.2) で定義される。

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, S) = \frac{1}{(\sqrt{2\pi})^D \sqrt{|S|}} \exp\left\{-\frac{1}{2}{}^t(\boldsymbol{x}-\boldsymbol{\mu})\ S^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\}. \tag{3.2}$$

母数 $\{w_k, \boldsymbol{\mu}_k, S_k\}$ の値を推定する前に、標本 $\{\boldsymbol{x}_n\}$ を K 個の部分集合に分類する問題を第 3.1 節で検討する。

3.1　クラスタリング

相互に近接した 2 点が同じ部分集合に配属されるように標本 $\{\boldsymbol{x}_n\}$ を分割する操作を**クラスタリング**と呼ぶ。直感的には、最適な部分集合 C_k は、重心 $\boldsymbol{\mu}_k$ と内部の座標 $\forall \boldsymbol{x} \in C_k$ との距離の総和を最小化する筈である。

$$\mathcal{D} = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2. \tag{3.3}$$

ただし、z_{nk} は点 $\boldsymbol{x}_n$ の所属を表す変数で、式 (3.4) に定義する。z_{nk} は観測 $\{\boldsymbol{x}_n\}$ の埒外の**潜在変数**である。

$$\hat{z}_{nk} = \begin{cases} 1 & \text{if } \boldsymbol{x}_n \in C_k \\ 0 & \text{if } \boldsymbol{x}_n \notin C_k. \end{cases} \tag{3.4}$$

最適な $\{z_{nk}\}$ と $\{\boldsymbol{\mu}_k\}$ は、反復法で求める。まず $\{\boldsymbol{\mu}_k\}$ を乱数で初期化し、次に式 (3.5) で $\{z_{nk}\}$ を更新する。

$$\hat{z}_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_k \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2 \\ 0 & \text{if } k \neq \arg\min_k \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2. \end{cases} \tag{3.5}$$

次に、式 (3.6) で $\{\boldsymbol{\mu}_k\}$ を更新する。以後、式 (3.5) の操作と式 (3.6) の操作を交互に反復し、収束解を得る。

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{N_k}\sum_{n=1}^{N} z_{nk}\boldsymbol{x}_n \text{ where } N_k = \sum_{n=1}^{N} z_{nk} \leftarrow \frac{\partial \mathcal{D}}{\partial \boldsymbol{\mu}_k} = 2\sum_{n=1}^{N} z_{nk}(\boldsymbol{x}_n - \boldsymbol{\mu}_k) = 0. \tag{3.6}$$

式 (3.5)(3.6) の反復で $\{C_k\}$ を求める手法を k-means と呼ぶ。下記の `Kmeans` クラスは k-means を実装する。

kmeans.scala

```
class Kmeans(X: Seq[Seq[Double]], K: Int, d: (Seq[Double],Seq[Double])=>Double) {
  val M = Array.fill(K, X.map(_.size).min)(Math.random)
```

標本 $\{\boldsymbol{x}_n\}$ は引数 `X` に渡す。引数 `d` は距離関数であり、通常は式 (3.3) に従って、下記の 2 乗関数を与える。

kmeans.scala

```
  def quad(a: Seq[Double], b: Seq[Double]) = (a,b).zipped.map(_-_).map(d=>d*d).sum
```

下記の `apply` メソッドは、指定した座標 $\boldsymbol{x}$ に対し、式 (3.5) に従って、至近の部分集合 C_k の番号 k を返す。

kmeans.scala

```
  def apply(x: Seq[Double]) = M.map(d(_,x)).zipWithIndex.minBy(_._1)._2
```

下記の `estep` メソッドは、標本 $\{\boldsymbol{x}_n\}$ を部分集合 $\{C_k\}$ に分配し、式 (3.6) に従って、重心 $\{\boldsymbol{\mu}_k\}$ を計算する。

kmeans.scala

```
  def estep = X.groupBy(apply).values.map(c=>c.transpose.map(_.sum / c.size))
```

最後に、式 (3.5)(3.6) の反復を実装する。回数は固定せずに、距離 $\mathcal{D}$ の収束を終了条件にすると完璧である。

kmeans.scala

```
  for(step<-1 to 100) (estep, M).zipped.foreach(_.copyToArray(_))
}
```

Fig. 3.2 は、$K=2$ の混合正規分布を `Kmeans` クラスで分割した結果で、特に黒縁の 2 点は重心 $\{\boldsymbol{\mu}_k\}$ を表す。

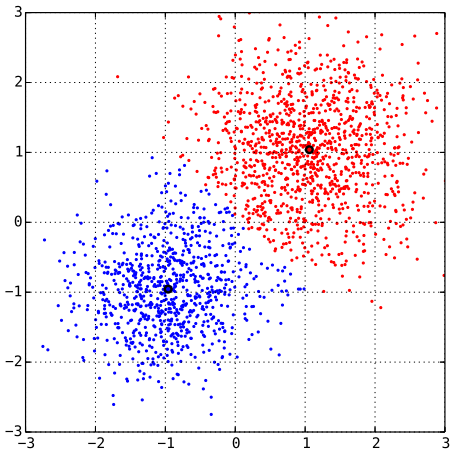


Fig. 3.2: k-means clustering on Gaussian mixture model.

概してクラスタリングは、標本 $\{\boldsymbol{x}_n\}$ が入力 $\boldsymbol{x}$ に対する出力を明示しない点で、教師なし学習に分類できる。

3.2 期待値最大化法

後述の通り、k-means は第 3.2 節で学ぶ期待値最大化法の特殊例であり、母数の初期値を求める際に役立つ。第 3.2 節では、変数 $\{z_{nk}\}$ を**確率変数**と見なし、重心 $\boldsymbol{\mu}_k$ に加えて加重 w_k と分散 S_k を推定する手順を学ぶ。

$$P(z_{nk}=1|\boldsymbol{x}_n)=\frac{w_k\mathcal{N}\left(\boldsymbol{x}_n|\boldsymbol{\mu}_k,S_k\right)}{\sum_{k=1}^{K}w_k\mathcal{N}\left(\boldsymbol{x}_n|\boldsymbol{\mu}_k,S_k\right)}=\gamma_{nk}. \tag{3.7}$$

式 (3.7) に定義した γ_{nk} は、観測 $\boldsymbol{x}_n$ を得た後の潜在変数 z_{nk} の確率分布を表すため、**事後確率**と呼ばれる。以後、母数 $\{w_k,\boldsymbol{\mu}_k,S_k\}$ の最尤推定の式を導出する。まず、母数の値の妥当性を担保する尤度 $\mathcal{L}$ を定義する。

$$\mathcal{L}\left(\{w_k,\boldsymbol{\mu}_k,S_k\}\right)=P(\{\boldsymbol{x}_n\}\,|\,\{w_k,\boldsymbol{\mu}_k,S_k\})=\prod_{n=1}^{N}P(\boldsymbol{x}_n|\,\{w_k,\boldsymbol{\mu}_k,S_k\}). \tag{3.8}$$

式 (3.8) に式 (3.1) を代入して、式 (3.9) を得る。対数は、微小値の積による指数部の**アンダーフロー**を防ぐ。

$$\log\mathcal{L}\left(\{w_k,\boldsymbol{\mu}_k,S_k\}\right)=\log\prod_{n=1}^{N}\sum_{k=1}^{K}w_k\mathcal{N}\left(\boldsymbol{x}_n|\boldsymbol{\mu}_k,S_k\right)=\sum_{n=1}^{N}\log\sum_{k=1}^{K}w_k\mathcal{N}\left(\boldsymbol{x}_n|\boldsymbol{\mu}_k,S_k\right). \tag{3.9}$$

以後、尤度 $\mathcal{L}$ の最大化を目指す。しかし、解析的な求解は困難である。式 (3.10) に $\boldsymbol{\mu}_k$ の偏微分の例を示す。

$$\frac{\partial}{\partial\boldsymbol{\mu}_k}\log\mathcal{L}=\frac{\partial}{\partial\boldsymbol{\mu}_k}\sum_{n=1}^{N}\log\sum_{k=1}^{K}w_k\mathcal{N}\left(\boldsymbol{x}_n|\boldsymbol{\mu}_k,S_k\right)=\sum_{n=1}^{N}\gamma_{nk}S_k^{-1}(\boldsymbol{x}_n-\boldsymbol{\mu}_k). \tag{3.10}$$

式 (3.10) の偏微分を 0 にする $\boldsymbol{\mu}_k$ が、平均 $\boldsymbol{\mu}_k$ の推定値 $\hat{\boldsymbol{\mu}_k}$ となる。分散 S_k の推定値 $\hat{S_k}$ も同様に計算する。

$$\left\{\begin{array}{l}\hat{\boldsymbol{\mu}_k}=\dfrac{1}{N_k}\displaystyle\sum_{n=1}^{N}\gamma_{nk}\boldsymbol{x}_n,\\ \hat{S_k}=\dfrac{1}{N_k}\displaystyle\sum_{n=1}^{N}\gamma_{nk}(\boldsymbol{x}_n-\boldsymbol{\mu}_k)\ {}^t(\boldsymbol{x}_n-\boldsymbol{\mu}_k)\ .\end{array}\right\}\ \text{where } N_k=\sum_{n=1}^{N}\gamma_{nk}. \tag{3.11}$$

分散 $\hat{S_k}$ は、変数 $\boldsymbol{x}$ の各次元の独立性を仮定する場合は、式 (3.12) で代用できる。$\circ$ は**要素ごとの積**を表す。

$$\hat{S_k}=\frac{1}{N_k}\left(\sum_{n=1}^{N}\gamma_{nk}\boldsymbol{x}_n^{\circ 2}\right)-\boldsymbol{\mu}_k^{\circ 2}. \tag{3.12}$$

加重 w_k の推定値 $\hat{w_k}$ は、式 (3.1) の制約条件を満たす必要があるので、**ラグランジュの未定乗数法**で求める。

$$\hat{w_k}=\frac{N_k}{N}. \tag{3.13}$$

事後確率 $\{\gamma_{nk}\}$ が求まれば母数も求まるが、$\{\gamma_{nk}\}$ の計算に $\{w_k,\boldsymbol{\mu}_k,S_k\}$ が必要であり、求解は困難である。そこで**補助関数法**を導入し、尤度 $\mathcal{L}$ の下限を与える補助関数 Q の最大化により、間接的に $\mathcal{L}$ を最大化する。

$$\log\mathcal{L}\left(\boldsymbol{\theta}\right)=\max_{\boldsymbol{\gamma}}Q(\boldsymbol{\gamma},\boldsymbol{\theta})\ \ \text{where } \boldsymbol{\gamma}=\{\gamma_{nk}\}\,,\ \boldsymbol{\theta}=\{w_k,\boldsymbol{\mu}_k,S_k\}\,. \tag{3.14}$$

補助関数 Q は、式 (3.15)(3.16) の更新を交互に反復することで最大化でき、最終的には有限な値に収束する。

$$\hat{\boldsymbol{\gamma}}^{t+1}=\arg\max_{\boldsymbol{\gamma}}Q(\boldsymbol{\gamma},\boldsymbol{\theta}), \tag{3.15}$$

$$\hat{\boldsymbol{\theta}}^{t+1}=\arg\max_{\boldsymbol{\theta}}Q(\boldsymbol{\gamma},\boldsymbol{\theta}). \tag{3.16}$$

最尤推定に補助関数法を併用し、式 (3.15)(3.16) の反復により最尤推定を行う手法を**期待値最大化法**と呼ぶ。ここで $f(x)$ を凸関数、$\{\gamma_n\}$ を正の実数列とすると、式 (3.17) が成立し、これを**イェンゼンの不等式**と呼ぶ。

$$\sum_{n=1}^{N} \gamma_n f(x_n) \geq f\left(\sum_{n=1}^{N} \gamma_n x_n\right) \text{ where } \sum_{n=1}^{N} \gamma_n = 1. \tag{3.17}$$

log が凹関数である点に注意して、式 (3.17) に式 (3.9) を当て嵌めると、式 (3.18) に示す補助関数 Q を得る。式 (3.15)(3.16) に関連して、補助関数 Q を最大化する $\hat{\boldsymbol{\gamma}}$ と $\hat{\boldsymbol{\theta}}$ を求めると、式 (3.7) と式 (3.11)(3.13) を得る。

$$\log \mathcal{L}(\boldsymbol{\theta}) \geq \sum_{n=1}^{N}\sum_{k=1}^{K} \gamma_{nk} \log \frac{w_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, S_k)}{\gamma_{nk}} = Q(\boldsymbol{\gamma}, \boldsymbol{\theta}). \tag{3.18}$$

期待値最大化法には別の解釈もある。仮に変数 $\{z_{nk}\}$ が観測可能ならば、対数尤度は式 (3.19) で定義される。実際には、変数 $\{z_{nk}\}$ の真の値は観測不可能なので、変数 $\{z_{nk}\}$ の事後分布に関して尤度の期待値を考える。

$$\log P(\{\boldsymbol{x}_n, z_{nk}\}|\boldsymbol{\theta}) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk} \log\{w_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, S_k)\}. \tag{3.19}$$

式 (3.15) や式 (3.7) で $\boldsymbol{\gamma}$ を求める操作は、式 (3.20) に示す**期待対数尤度**の最新値を求める操作と等価であり、式 (3.16) や式 (3.11)(3.13) で $\boldsymbol{\theta}$ を求める操作は、その期待対数尤度を極大点まで近付ける操作と等価である。

$$\mathop{\mathbf{E}}_{\boldsymbol{z}}[\log P(\{\boldsymbol{x}_n, z_{nk}\}|\boldsymbol{\theta})] = \sum_{n=1}^{N}\sum_{k=1}^{K} \gamma_{nk} \log\{w_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, S_k)\}. \tag{3.20}$$

これが期待値最大化法の名の由来であり、式 (3.15) を E-射影と呼び、式 (3.16) を M-射影と呼ぶ根拠である。余談だが、分散をスカラー行列 εE とすれば $\varepsilon \to \infty$ の極限で式 (3.21) が成立し、更に $\gamma_{nk} \to z_{nk}$ も成立する。

$$\lim_{\varepsilon\to 0} \varepsilon \log\{w\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \varepsilon E)\} = \lim_{\varepsilon\to 0}\left\{\varepsilon \log w - \varepsilon\frac{D}{2}\log(2\pi\varepsilon) - \frac{1}{2}\|\boldsymbol{x}-\boldsymbol{\mu}\|^2\right\} = -\frac{1}{2}\|\boldsymbol{x}-\boldsymbol{\mu}\|^2. \tag{3.21}$$

従って、式 (3.22) が成立し、更に、期待値最大化法による式 (3.22) の最大化は式 (3.3) の最小化に帰結する。

$$\epsilon \mathop{\mathbf{E}}_{\boldsymbol{z}}[\log P(\{\boldsymbol{x}_n, z_{nk}\}|\boldsymbol{\theta})] \to -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2 + C. \tag{3.22}$$

期待値最大化法を実装する前に K 個の正規分布 $\{\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, S_k)\}$ の混合分布を下記の `GMM` クラスに実装する。

ExpectationMaximization.scala

```
class GMM(val D: Int, val K: Int) {
  val W = Array.fill(K)(1.0 / K)
  val M = Array.fill(K, D)(math.random)
  val S = Array.fill(K, D)(math.random)
```

配列 `W` と `M` と `S` は、加重 $\{w_k\}$ と平均 $\{\boldsymbol{\mu}_k\}$ と分散 $\{S_k\}$ である。次の `apply` メソッドは式 (3.2) を計算する。

ExpectationMaximization.scala

```
  def apply(x: Seq[Double]) = for((w,m,s)<-(W,M,S).zipped) yield {
    val p = math.exp(-(x,m,s).zipped.map((x,m,s)=>(x-m)*(x-m)/s).sum/2)
    w * p / (math.pow(2 * math.Pi, .5 * m.size) * math.sqrt(s.product))
  }
}
```

下記の `EM` クラスは、期待値最大化法を実装する。標本 $\{\boldsymbol{x}_n\}$ は引数 `X` に、正規分布の個数は引数 `K` に渡す。

ExpectationMaximization.scala

```
class EM(X: Seq[Seq[Double]], K: Int) {
  val gmm = new GMM(X.map(_.size).min, K)
```

下記の apply メソッドは、事後確率 γ_k の最大値を与える k を返す。第 3.1 節の apply メソッドに相当する。

ExpectationMaximization.scala

```
  def apply(x: Seq[Double]) = gmm(x).toSeq.zipWithIndex.maxBy(_._1)._2
```

次に M-射影を実装する。式 (3.13)(3.11)(3.12) に従って母数 $\{w_k, \boldsymbol{\mu}_k, S_k\}$ を更新し、式 (3.9) の尤度を返す。

ExpectationMaximization.scala

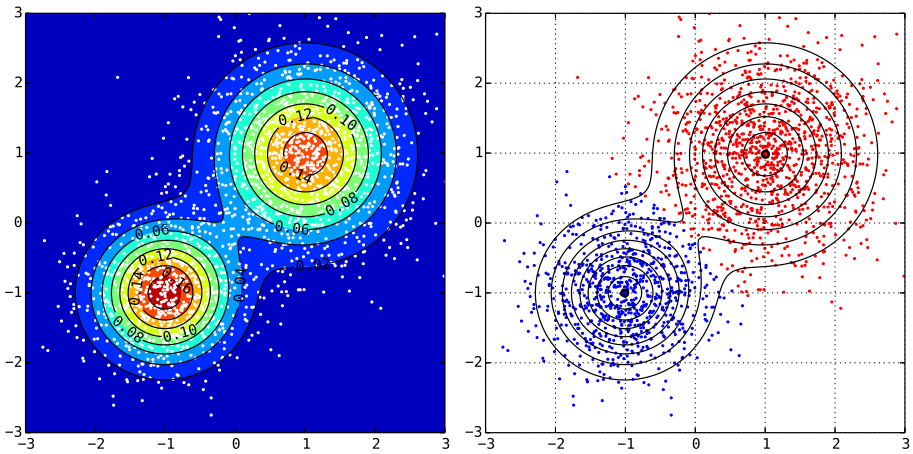
```
  def mstep(P: Seq[Seq[Double]]): Double = {
    for(k<-0 until K) gmm.W(k) = P(k).sum / X.size
    val m1 = P.map((_,X).zipped.map((p,x)=>x.map(x=>p*x*1)).transpose)
    val m2 = P.map((_,X).zipped.map((p,x)=>x.map(x=>p*x*x)).transpose)
    for(k<-0 until K; d<-0 until gmm.D) {
      gmm.M(k)(d) = m1(k)(d).sum / P(k).sum
      gmm.S(k)(d) = m2(k)(d).sum / P(k).sum - math.pow(gmm.M(k)(d), 2)
    }
    X.map(x=>math.log(gmm(x).sum)).sum
  }
```

mstep メソッドの引数は、標本 $\{\boldsymbol{x}_n\}$ の事後確率 $\{\gamma_{nk}\}$ である。これは、式 (3.7) の E-射影に従って求める。下記は、E-射影と M-射影を交互に反復する。回数を固定せずに、尤度の収束を目安にすると実用的である。

ExpectationMaximization.scala

```
  for(step <- 1 to 100) mstep(X.map(gmm(_)).map(p=>p.map(_/p.sum)).transpose)
}
```

Fig. 3.3 は、Fig. 3.2 と同じ標本を EM クラスで学習した結果で、等高線は、混合正規分布の密度関数を示す。特に、(a) は密度関数の**ヒートマップ**を表す。また、(b) は混合正規分布によるクラスタリングの結果である。



(a) density map. (b) cluster map.

Fig. 3.3: expectation maximization on a Gaussian mixture model.

Fig. 3.2 に比べ、正規分布が接する谷間に位置する標本は、正規分布の等高線を正しく反映した分類になる。

第4章　潜在的ディリクレ配分法

第 4 章では、観測データの生成過程を確率変数の因果関係などの連鎖で表現する**グラフィカルモデル**を学ぶ。自然言語処理に焦点を当て、百科事典の記事を訓練データに使う。まず、XML の**ダンプデータ**を入手する。

```
$ wget https://dumps.wikimedia.org/jawiki/latest/jawiki-latest-abstract.xml
```

次に、Scala で XML 文書を操作する API を利用して、記事の題名と本文を抽出し、適当な配列に格納する。

MorphologicalAnalysis.scala

```
val docs = scala.xml.XML.loadFile("jawiki-latest-abstract.xml") \\ "doc"
val data = Map(docs.map(d => (d\"title").text->(d\"abstract").text): _*)
```

記事には、文章を単語列に分解する**形態素解析**を予め施す。形態素解析器は、Lucene GoSen が便利だろう。

MorphologicalAnalysis.scala

```
val gosen = net.java.sen.SenFactory.getStringTagger(null)
val words = gosen.analyze(data("Wikipedia: Scala"), new ArrayList[Token]())
```

助詞や助動詞や接続詞など**機能語**は、機械学習の際には無用のため、形態素解析の段階での排除を推奨する。式 (4.1) に示す *tf-idf* を指標に、特定の記事 d で高頻度だが、他の記事で低頻度な単語 t_d を残す方法もある。

$$\mathrm{tf\,idf}(t_d \in d, d \in D) = \mathrm{tf}(t_d, d)\mathrm{idf}(t_d, D) \quad \text{where} \quad \begin{cases} \mathrm{tf}(t_d, d) = \log\left(1 + \dfrac{N_{t_d d}}{\sum_t N_{td}}\right) \\ \mathrm{idf}(t, D) = \log \dfrac{|D|}{|\{d : t \in d\}|} \end{cases} \tag{4.1}$$

ただし、N_{td} は文書 d に単語 t が出現する回数である。また、式 (4.1) の対数は、tf と idf の変動を抑制する。

4.1　単純ベイズ分類器

第 4.1 節で学ぶ**単純ベイズ分類器**は、単語 w_n の列である文書 d が、話題 c から生成された確率を推定する。

$$P(d|c) = P(w_1, \ldots, w_{N_d}|c) = \prod_{n=1}^{N_d} P(w_n|c). \tag{4.2}$$

式 (4.2) の $P(d|c)$ は、**観測変数**たる文書 d が潜在変数たる話題 c から生成された場合の、仮説の尤度を表す。単語の**共起**を考慮すると式 (4.3) になるが、寧ろ独立性に拘る式 (4.2) は、この分類器の単純たる理由である。

$$P(w_1, \ldots, w_{N_d}|c) = \prod_{n=1}^{N_d} P(w_n|c, w_1, ..., w_{n-1}). \tag{4.3}$$

文書 d を生成した話題 $\hat{c}_d$ を推測する方法を検討する。直感的には、話題 $\hat{c}_d$ は確率 $P(c|d)$ の最大値を与える。

$$\hat{c}_d = \arg\max_{c \in C} P(c|d). \tag{4.4}$$

確率 $P(d)$ が話題 c に対し独立である点に留意して、式 (4.4) に**ベイズの定理**を適用すると、式 (4.5) を得る。

$$\hat{c}_d = \arg\max_{c\in C} \frac{P(d|c)P(c)}{P(d)} = \arg\max_{c} P(d|c)P(c). \tag{4.5}$$

独立性を仮定した式 (4.2) を式 (4.5) に代入すると、式 (4.6) を得る。話題を判定する際は、式 (4.6) を使う。

$$\hat{c}_d = \arg\max_{c\in C} P(c) \prod_{n=1}^{N_d} P(w_n|c). \tag{4.6}$$

なお、確率 $P(w|c)$ は未知の単語 w に対して 0 になる。対策として、式 (4.7) の**ラプラス平滑化**で補正する。式 (4.7) の確率 $P(w)$ は未知語 w の**事前確率**を、確率 $P(w|c)$ は訓練データを学習した後の事後確率を表す。

$$P(w|c) = \frac{N_{wc}+1}{\sum_{w\in V}(N_{wc}+1)} \leftarrow w \sim P(w) = \mathrm{Uniform}(V) = \frac{1}{|V|}. \tag{4.7}$$

下記の `NaiveBayes` クラスは、観測 d を得て潜在変数 c の事後確率 $P(d|c)$ を最大化する話題 $\hat{c}_d$ を推定する。型引数 `W` と `C` は単語と話題を表す。引数 `docs` と `cats` には、文書 $\{d_i\}$ と話題 $\{c_i\}$ を同じ順番に並べて渡す。

NaiveBayes.scala

```
class NaiveBayes[W,C](docs: Seq[Seq[W]], cats: Seq[C]) {
  val N = scala.collection.mutable.Map[(W,C),Double]().withDefaultValue(0)
```

下記の配列 `P` は、話題 c の事前確率 $P(c)$ を保持する。単に、訓練データで話題 $\{c\}$ が出現した頻度である。

NaiveBayes.scala

```
  val P = cats.groupBy(c=>c).map{case (c,s)=>c->s.size.toDouble/docs.size}
```

次に、単語 w と話題 c の共起の回数 N_{wc} を配列 `N` に格納する。配列 `N` は未知語 w に対して $N_{wc}=0$ を返す。

NaiveBayes.scala

```
  for((d,c)<-(docs,cats).zipped; w<-d) N(w,c) += 1
```

`Pwc` メソッドは、単語 w を観測した場合の、その原因たる話題 c の尤度 $P(w|c)$ を式 (4.7) に従って計算する。

NaiveBayes.scala

```
  def Pwc(w: W, c: C) = (N(w,c)+1) / docs.flatten.distinct.map(N(_,c)+1).sum
```

`Pcd` メソッドは、文書 d を観測した際の、その原因たる話題 c の事後確率 $P(c|d)$ を式 (4.6) により計算する。

NaiveBayes.scala

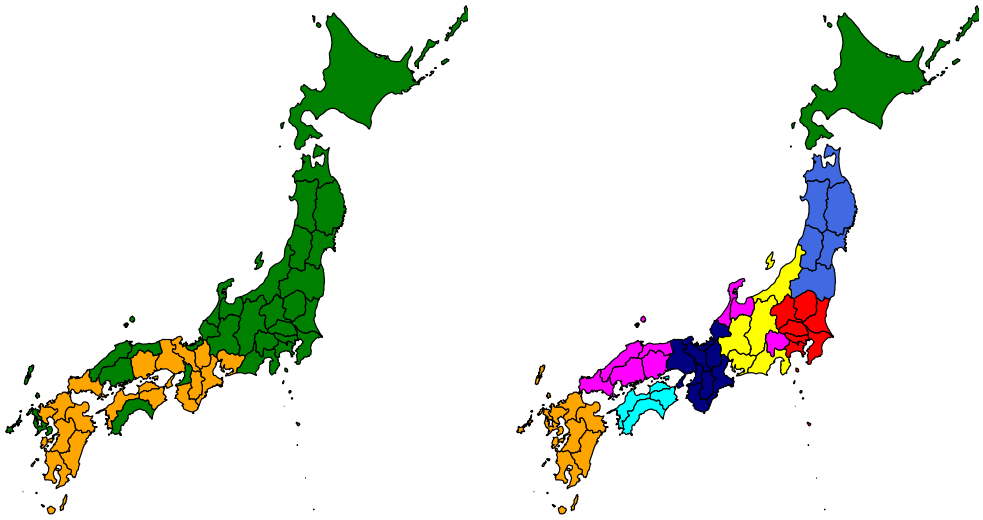
```
  def Pcd(c: C, d: Seq[W]) = math.log(P(c)) + d.map(w=>math.log(Pwc(w,c))).sum
```

最後に `apply` メソッドを定義する。未知の文書 d に対し、事後確率 $P(c|d)$ を最大化する話題 $\hat{c}_d$ を推定する。

NaiveBayes.scala

```
  def apply(d: Seq[W]) = cats.distinct.maxBy(Pcd(_, d))
}
```

Fig. 4.1(a) は、固有名詞を特徴量として東日本と西日本の記事を学習し、46 都道府県を分類した結果である。同様に、北海道と東北、関東と中部、近畿と中国、四国に九州の 8 地方に分類した結果を Fig. 4.1(b) に示す。



(a) 2-regional division. (b) 8-regional division.

Fig. 4.1: Japanese map division into regions based on classification of Wikipedia pages.

4.2 単語の話題の推定

第 4.2 節で学ぶ**潜在的ディリクレ配分法**は、話題に応じて単語の意味が変化する複雑な言語モデルを扱える。実際の自然言語の文では、短文でも複数の話題に言及し得るので、単語と意味を区別して考える必要がある。

例文 *I wrote a Java program drinking a cup of Java coffee.*

Fig. 4.2 に示す通り、複雑な言語モデルは単語 w や話題 z を始め、複数の確率変数の依存関係で表現される。

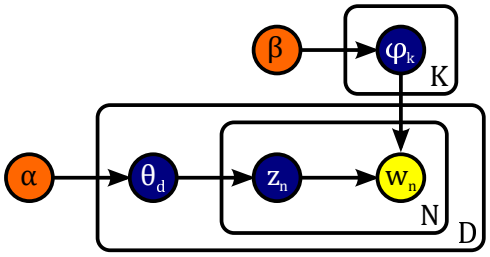


Fig. 4.2: latent Dirichlet allocation model.

文書 $d \in D$ に出現する n 番目の単語 w_{dn} の話題を変数 z_{dn} で表す。両変数は式 (4.8)(4.9) の**多項分布**に従う。ただし、V は語彙量を、K は話題の数を表し、$N_v \in \{0,1\}$ と $N_k \in \{0,1\}$ は単語 v と話題 k の出現を表す。

$$w_{dn} \sim P(w|z) = \prod_{v=1}^{V} \phi_{zv}^{N_v} = \mathrm{Mul}\,(\boldsymbol{\phi}_z) \quad \text{where} \quad \sum_{v=1}^{V} N_v = 1, \tag{4.8}$$

$$z_{dn} \sim P(z|d) = \prod_{k=1}^{K} \theta_{dk}^{N_k} = \mathrm{Mul}\,(\boldsymbol{\theta}_d) \quad \text{where} \quad \sum_{k=1}^{K} N_k = 1. \tag{4.9}$$

ϕ_{zv} は話題 z が単語 v を生起する確率 $P(v|z)$ を表し、θ_{dk} は文書 d が話題 k を生起する確率 $P(k|d)$ を表す。単語 w と文書 d の結合確率は式 (4.10) で与える。話題 z は**潜在変数**で観測の埒外にあるので、周辺化する。

$$P(w,d) = P(d) \sum_{z=1}^{K} P(w|z)P(z|d) = \int_z \phi_z(w)\theta_d(z)dz. \tag{4.10}$$

観測された文書 d を的確に表す母数 ϕ_z や母数 θ_d を推定する場合は、式 (4.11) の尤度関数の極値を探索する。

$$\mathcal{L}(\phi_z, \theta_d) = P(\boldsymbol{w}_d|\phi_z, \theta_d) = \prod_{n=1}^{N_d} \int_z \phi_z(w_{dn})\theta_d(z)dz. \tag{4.11}$$

ただし、母数 θ_d の個数は文書の量に比例し、膨大な数になるため、単なる最尤推定では過学習を頻発する。対策として**ベイズ推定**を導入し、文書 d を観測した後の母数 θ_d や母数 ϕ_z の分布を**事後分布**として計算する。

$$P(\phi_z, \theta_d|\boldsymbol{w}_d) = \frac{P(\boldsymbol{w}_d|\phi_z, \theta_d)P(\phi_z, \theta_d)}{P(\boldsymbol{w}_d)} \propto P(\boldsymbol{w}_d|\phi_z, \theta_d)P(\phi_z, \theta_d). \tag{4.12}$$

式 (4.12) の $P(\phi_z, \theta_d)$ を**事前分布**と呼び、文書 d を観測する前に予想された母数 ϕ_z や母数 θ_d の分布を表し、過学習を抑える働きを持つ。事後分布は尤度と事前分布の積に比例し、最適な母数は事後確率を最大化する。

$$\begin{aligned} \hat{\phi}_z &= \arg\max_{\phi_z} P(\phi_z, \theta_d|\boldsymbol{w}_d), \\ \hat{\theta}_d &= \arg\max_{\theta_d} P(\phi_z, \theta_d|\boldsymbol{w}_d). \end{aligned} \tag{4.13}$$

なお、式 (4.12) は、1 件の文書を観測する場合を想定した。文書が複数ある場合には、式 (4.14) を適用する。

$$P(\phi_z, \theta_1, ..., \theta_D|\boldsymbol{w}) \propto P(\phi_z, \theta_1, ..., \theta_D) \prod_{d=1}^{D} P(\boldsymbol{w}_d|\phi_z, \theta_d). \tag{4.14}$$

式 (4.14) は、文書 d_t を観測した後の事後分布を事前分布として次の文書 d_{t+1} を学習する連鎖の様子を表す。連鎖を容易にする目的から、事前分布は式 (4.15) の**ディリクレ分布**で定義する。$\Gamma(z)$ は**ガンマ関数**である。

$$\mathrm{Dir}(\boldsymbol{\theta}|\boldsymbol{\alpha}) = \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \quad \text{where } \Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt. \tag{4.15}$$

式 (4.8)(4.9) より、式 (4.11) の尤度は多項分布を描くが、式 (4.16) より、事後確率もディリクレ分布に従う。

$$\mathrm{Dir}(\boldsymbol{\theta}|\boldsymbol{n} + \boldsymbol{\alpha}) \propto \mathrm{Mul}(\boldsymbol{n}|\boldsymbol{\theta})\,\mathrm{Dir}(\boldsymbol{\theta}|\boldsymbol{\alpha}). \tag{4.16}$$

式 (4.16) の性質により、事前確率は事後確率と共通の確率密度関数で定義でき、これを**共役事前分布**と呼ぶ。

$$\begin{aligned} \phi_k &\sim \mathrm{Dir}(\phi|\boldsymbol{\beta}), \\ \theta_d &\sim \mathrm{Dir}(\theta|\boldsymbol{\alpha}). \end{aligned} \tag{4.17}$$

母数 ϕ_k と母数 θ_d の事前分布を式 (4.17) に設定し、式 (4.12) のベイズ推定に組み込むと、式 (4.18) を得る。

$$P(\boldsymbol{\phi}, \boldsymbol{\theta}|\boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \propto \prod_{k=1}^{K} \mathrm{Dir}(\phi_k|\boldsymbol{\beta}) \prod_{d=1}^{D} \mathrm{Dir}(\theta_d|\boldsymbol{\alpha}) \prod_{n=1}^{N_d} \int_z \phi_z(w_{dn})\theta_d(z)dz. \tag{4.18}$$

式 (4.18) の事後確率を最大化する点を微分により求めれば、母数 ϕ_k や母数 θ_d の具体的な数値が得られるが、微分は困難である。そこで、話題 z_{dn} の標本を近似的に生成し、式 (4.19) で母数を計算する方針を検討する。

$$\begin{aligned} \hat{\phi}_{kv} &= \frac{N_{kv} + \beta_v}{\sum_{v=1}^{V} (N_{kv} + \beta_v)}, \\ \hat{\theta}_{dk} &= \frac{N_{dk} + \alpha_k}{\sum_{k=1}^{K} (N_{dk} + \alpha_k)}. \end{aligned} \tag{4.19}$$

ただし、N_{kv} は話題 k で単語 v が出現した回数を表し、N_{dk} は文書 d で話題 k の単語が出現した回数を表す。時刻 t の標本 $\{z_{dn}^t\}$ は、式 (4.20) の**提案分布**に従う乱数により生成する。これを**ギブスサンプリング**と呼ぶ。

$$\forall z_{dn}^t \sim P(z_{dn}|z_{11}^t, ..., z_{dn-1}^t, z_{dn+1}^{t-1}, ..., z_{DN_D}^{t-1}). \tag{4.20}$$

変数 ϕ_z と変数 θ_d も潜在変数なので、本来は標本に含むべきであるが、式 (4.21) の周辺化により除去できる。

$$P(\boldsymbol{z}, \boldsymbol{w}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \left(\prod_{k=1}^{K} \int \mathrm{Dir}\left(\phi_k|\boldsymbol{\beta}\right) \prod_{d=1}^{D} \prod_{n=1}^{N_d} \phi_{z_{dn}}(w_{dn})\, d\phi_k \right) \left(\prod_{d=1}^{D} \int \mathrm{Dir}\left(\theta_d|\boldsymbol{\alpha}\right) \prod_{n=1}^{N_d} \theta_d(z_{dn})\, d\theta_d \right). \tag{4.21}$$

詳細は省略するが、**ベータ関数**の積分が出現する点に注目しつつ、式 (4.21) を変形すると、式 (4.22) を得る。

$$P(\boldsymbol{z}, \boldsymbol{w}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \prod_{k=1}^{K} \left\{ \frac{\Gamma\left(\sum_{v=1}^{V} \beta_v\right)}{\prod_{v=1}^{V} \Gamma\left(\beta_v\right)} \frac{\prod_{v=1}^{V} \Gamma\left(N_{kv} + \beta_v\right)}{\Gamma\left(N_k + \sum_{v=1}^{V} \beta_v\right)} \right\} \prod_{d=1}^{D} \left\{ \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma\left(\alpha_k\right)} \frac{\prod_{k=1}^{K} \Gamma\left(N_{dk} + \alpha_k\right)}{\Gamma\left(N_d + \sum_{k=1}^{K} \alpha_k\right)} \right\}. \tag{4.22}$$

式 (4.22) で単語 w_{dn} と話題 z_{dn} に依存する部分を式 (4.20) の条件付き分布に代入すると、式 (4.23) を得る。ただし、式 (4.23) で、$\boldsymbol{w}^{\not{dn}}$ は単語 w_{dn} を除外した単語の列であり、$\boldsymbol{z}^{\not{dn}}$ は $\boldsymbol{w}^{\not{dn}}$ に対応する話題の列である。

$$P(z_{dn}|\boldsymbol{z}^{\not{dn}}) \propto P(z_{dn} = k, w_{dn} = v, \boldsymbol{z}^{\not{dn}}, \boldsymbol{w}^{\not{dn}}|\boldsymbol{\alpha}, \boldsymbol{\beta}) \propto \frac{N_{kv}^{\not{dn}} + \beta_v}{N_k^{\not{dn}} + \sum_{v=1}^{V} \beta_v} \frac{N_{dk}^{\not{dn}} + \alpha_k}{N_d^{\not{dn}} + \sum_{k=1}^{K} \alpha_k}. \tag{4.23}$$

式 (4.23) の提案分布に従う標本生成を何度も反復すれば、最終的に、標本 $\boldsymbol{z}$ は真の分布に収束する筈である。下記の `LDA` クラスは、潜在的ディリクレ配分法を実装する。引数 `docs` に文書の題名と単語列の配列を渡す。

LatentDirichletAllocation.scala

```
class LDA[W,D](docs: Map[D, Seq[W]], K: Int, a: Double = 0.1, b: Double = 0.01) {
```

引数 `K` は話題の個数で、引数 `a` と `b` は $\forall a_k = a, \forall b_v = b$ を仮定した**対称ディリクレ分布**の母数 α と β である。`LDA` クラス内に、単語 w_{dn} を包む `Word` クラスを実装する。乱数で初期化される変数 `z` は、話題 z_{dn} を表す。

LatentDirichletAllocation.scala

```
  case class Word(v: W, var z: Int = util.Random.nextInt(K))
```

配列 `W` は集合 $\{w_{dn}, z_{dn}\}$ に相当する。文書 d と位置 n を索引として `Word` クラスのインスタンスを管理する。

LatentDirichletAllocation.scala

```
  val W = docs.mapValues(_.map(Word(_)).toArray)
```

配列 `V` は語彙 V の要素 $v \in V$ を索引として集合 $\{w_{dn}, z_{dn}\}$ を管理する。変数 N_{kv} の値を計算する際に使う。

LatentDirichletAllocation.scala

```
  val V = W.flatMap(_._2).groupBy(_.v)
```

下記の `Ndk` メソッドは式 (4.19) の変数 N_{dk} に相当し、文書 d に含まれる話題 k の単語の出現回数を数える。

LatentDirichletAllocation.scala

```
  def Ndk(d: D) = 0.until(K).map(k=>W(d).count(_.z==k) + a)
```

Nkv メソッドは式 (4.19) の変数 N_{kv} に相当し、単語の列 $\{w_{dn}\}$ の部分集合 $\{w_{dn}=v, z_{dn}=k\}$ の濃度を返す。

LatentDirichletAllocation.scala

```
  def Nkv(v: W) = 0.until(K).map(k=>V(v).count(_.z==k) + b)
```

配列 NkV は N_{kv} を V 全体に渡って合計した値である。実行効率の都合から、関数ではなく配列で実装した。

LatentDirichletAllocation.scala

```
  val NkV = V.keys.map(Nkv).reduce((_,_).zipped.map(_+_)).toArray
```

次に、式 (4.20) に従う乱数で標本 $\{z^t_{dn}\}$ を更新する処理を記述する。この処理は、収束まで何度も繰り返す。

LatentDirichletAllocation.scala

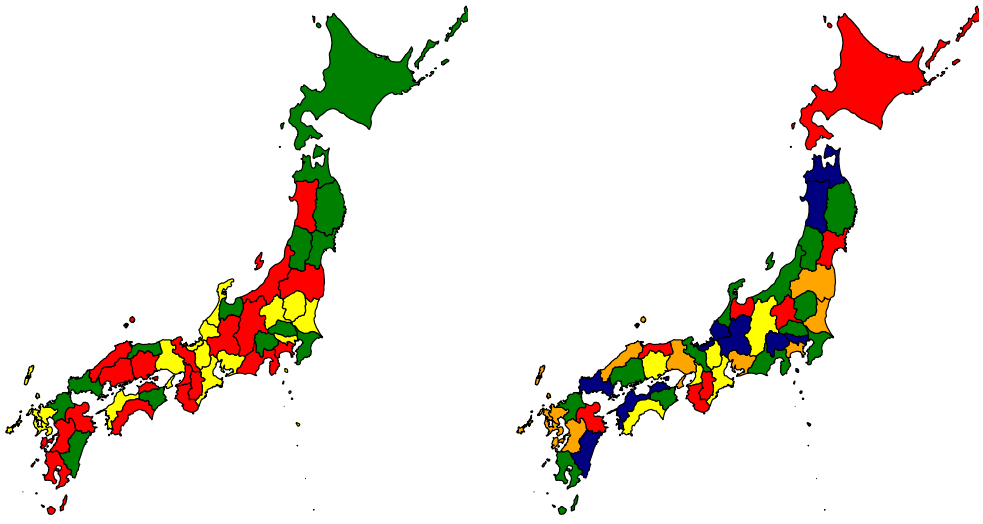
```
  for(step<-1 to 1000; d<-docs.keys; (w,n)<-docs(d).zipWithIndex) {
    NkV(W(d)(n).z) -= 1
    W(d)(n).z = -1
    val S = (Nkv(w), Ndk(d), NkV).zipped.map(_*_/_).scan(.0)(_+_)
    val r = util.Random.nextDouble * S.last
    W(d)(n).z = S.tail.indexWhere(_ >= r)
    NkV(W(d)(n).z) += 1
  }
```

apply メソッドは、文書 d で支配的な話題を探す。未知の文書には適用できず、専らクラスタリングに使う。

LatentDirichletAllocation.scala

```
  def apply(d: D) = 0.until(K).maxBy(Ndk(d)(_))
}
```

固有名詞を特徴量として 46 都道府県の記事を LDA クラスで学習し、色を塗り分けた結果を Fig. 4.3 に示す。



(a) 3-regional division.　　(b) 5-regional division.

Fig. 4.3: Japanese map division into regions based on clustering of Wikipedia pages.

教師なし学習であるため第 4.1 節と比べて分類結果の評価は難しく、変数 N_{kv} を詳細に解析する必要がある。

第5章　ニューラルネットワーク

ニューラルネットワークは神経系を模倣した計算模型であり、神経細胞に相当する単位を**ニューロン**と呼ぶ。

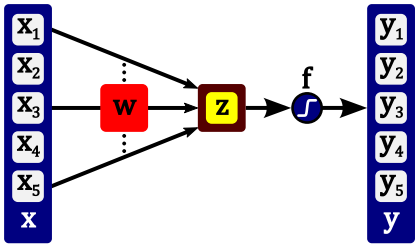


Fig. 5.1: a neuron.

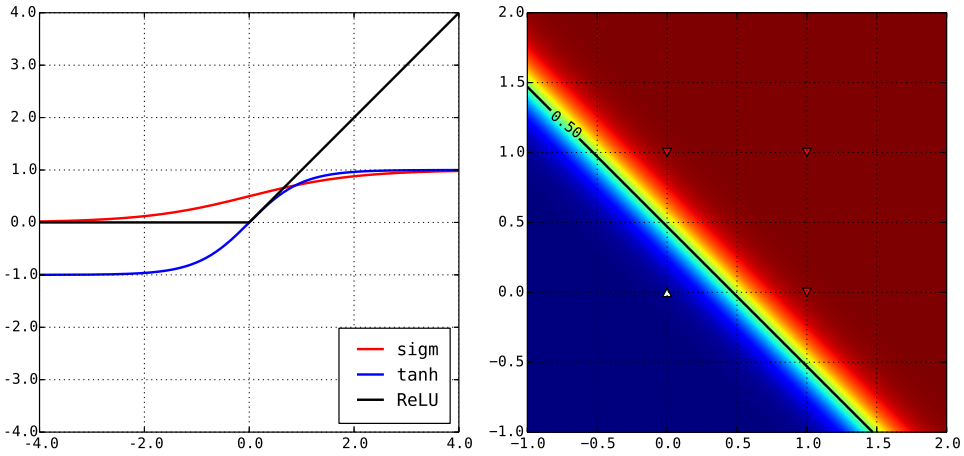
細胞は D 本の樹状突起で信号 $\boldsymbol{x}$ を受容し、加重 $\boldsymbol{w}$ で合計した後、**活性化関数** f の値を軸索末端で放出する。

$$y = f(z) = f(\boldsymbol{w} \cdot \boldsymbol{x}) = f\left(\sum_{d=1}^{D} w_d x_d\right). \tag{5.1}$$

関数 f は、神経細胞が興奮する閾値に相当する。f が恒等関数 $f_{\mathrm{id}}(z)=z$ の場合、式 (5.1) は線型回帰になる。他方、関数 f を階段関数や式 (5.2) の**シグモイド関数**で実装すると、式 (5.1) は $y \in \{0,1\}$ の 2 値分類になる。

$$f_{\mathrm{sigm}}(z) = \frac{1}{1+e^{-z}} = \frac{1}{2}\tanh\frac{z}{2} + \frac{1}{2}. \tag{5.2}$$

他にも Fig. 5.2(a) に示す tanh 関数や ReLU 関数が存在し、回帰やクラス分類など用途に応じて選択できる。Fig. 5.2(b) は、論理和を $f = f_{\mathrm{sigm}}$ の分類器で学習した結果である。直線 $f(\boldsymbol{x})=0.5$ はクラスの境界を表す。



(a) activation functions.　　(b) OR learned by neuron.

Fig. 5.2: neuron mechanism.

分類器としてのニューロンは線型分類器と呼ばれ、境界が超平面になる**線型分離可能**な問題のみ学習できる。

5.1 誤差逆伝播法

Fig. 5.3 は、ニューロンの多層化により非線型な回帰問題や分類問題に対応した**多層パーセプトロン**である。左端で信号 $\boldsymbol{x}_1$ を受容する層を**入力層**、右端で信号 $\boldsymbol{x}_3$ を出力する層を**出力層**と呼び、その間を**隠れ層**と呼ぶ。

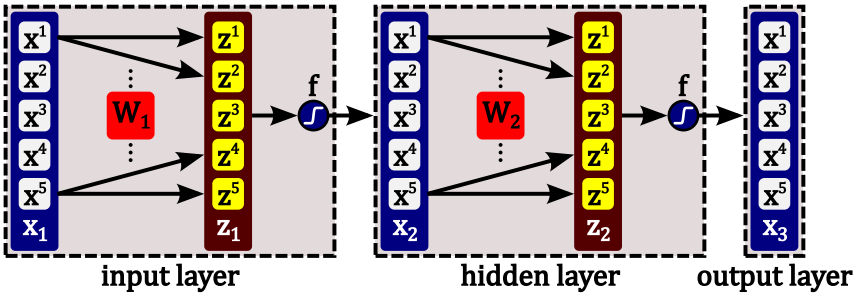


Fig. 5.3: multi-layer perceptron.

信号 $\boldsymbol{x}$ は層を経る度に非線型に変換されるため、3 層あれば任意の**滑らかな関数**を任意の精度で近似できる。入力層と隠れ層の活性化関数を f とし、第 m 層の加重を W_m と表記する。出力 $\boldsymbol{y}$ は式 (5.3) で定義される。

$$\boldsymbol{y} = \boldsymbol{x}_3 = f(\boldsymbol{z}_2) = f(W_2\boldsymbol{x}_2) = f(W_2 f(\boldsymbol{z}_1)) = f(W_2 f(W_1\boldsymbol{x}_1)). \tag{5.3}$$

式 (5.3) の学習は、出力 $\{\boldsymbol{y}\}$ と正解 $\{\boldsymbol{t}\}$ の誤差を表す**損失関数** E が最小になる加重 W の探索を通じて行う。関数 E は、出力層の直前の関数 f に応じて選ぶべきだが、式 (5.4) の 2 乗誤差は、特定の f に拠らず使える。

$$E_{\mathrm{sq}}(\boldsymbol{y} \sim q(\boldsymbol{y}), \boldsymbol{t} \sim p(\boldsymbol{t})) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{t}\|^2 \quad \text{where } \|\boldsymbol{a}\| = \sqrt{|a_1|^2 + \cdots + |a_d|^2}. \tag{5.4}$$

式 (5.5) の**交差エントロピー** E_{CE} は分布 p と q の差を表す Kullback–Leibler 情報量 $D(p\|q)$ の上限を与える。誤差 E_{CE} の最小化は $D(p\|q)$ の直接的な最小化に比べ、誤差逆伝播法に馴染む。詳細は第 5.3 節で解説する。

$$E_{\mathrm{CE}}(p,q) = -\int p(\boldsymbol{y}) \log q(\boldsymbol{y}) d\boldsymbol{y} = -\int p(\boldsymbol{y}) \left\{ \log p(\boldsymbol{y}) - \log \frac{p(\boldsymbol{y})}{q(\boldsymbol{y})} \right\} d\boldsymbol{y} = H(p) + D(p\|q)\,. \tag{5.5}$$

第 m 層の第 i 成分から第 $m{+}1$ 層の第 j 成分への経路 $i \to j$ の加重 w_m^{ij} は、式 (5.6) の勾配法で最適化できる。係数 η は学習率である。また、変数 x_m^i は第 m 層の入力 $\boldsymbol{x}_m$ の第 i 成分で z_m^j は $W_m\boldsymbol{x}_m$ の第 j 成分である。

$$w'^{ij}_m = w^{ij}_m - \eta \frac{\partial E}{\partial w^{ij}_m} = w^{ij}_m - \eta \frac{\partial z^j_m}{\partial w^{ij}_m} \frac{\partial x^j_{m+1}}{\partial z^j_m} \frac{\partial E}{\partial x^j_{m+1}} = w^{ij}_m - \eta x^i_m \frac{\partial f}{\partial z^j_m}(z^j_m) \frac{\partial E}{\partial x^j_{m+1}}. \tag{5.6}$$

各層で損失関数 E の導関数が必要だが、式 (5.7) の漸化式を通じ、出力層から入力層に向けて逆伝播できる。

$$\frac{\partial E}{\partial x^i_m} = \sum_{j=1}^{J} \frac{\partial z^j_m}{\partial x^i_m} \frac{\partial x^j_{m+1}}{\partial z^j_m} \frac{\partial E}{\partial x^j_{m+1}} = \sum_{j=1}^{J} w^{ij}_m \frac{\partial f}{\partial z^j_m}(z^j_m) \frac{\partial E}{\partial x^j_{m+1}}. \tag{5.7}$$

なお、活性化関数 f は式 (5.6)(5.7) の適用を助ける目的で、ReLU 関数を除き**微分可能**な関数で設計される。

$$\frac{\partial f_{\mathrm{sigm}}}{\partial z^j_m}(z^j_m) = \frac{e^{-z^j_m}}{(1+e^{-z^j_m})^2} = x^j_{m+1}(1 - x^j_{m+1}). \tag{5.8}$$

損失関数 E の逆伝播による学習を**誤差逆伝播法**と呼ぶ。出力層の直前の E_{sq} の導関数は、式 (5.9) で求まる。

$$\frac{\partial E_{\mathrm{sq}}}{\partial x^j_3}(\boldsymbol{x}_3, \boldsymbol{t}) = \frac{\partial}{\partial x^j_3} \frac{1}{2}\|\boldsymbol{x}_3 - \boldsymbol{t}\|^2 = x^j_3 - t^j. \tag{5.9}$$

以上の議論に基づき、ニューラルネットワークと誤差逆伝播法を実装する。まず、活性化関数 f を定義する。

NeuralNetwork.scala

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

Active を継承した Sigmoid クラスは関数 f_{sigm} を表す。式 (5.2)(5.8) を fp と bp の各メソッドで実装する。

NeuralNetwork.scala

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

Fig. 5.3 の入力層と隠れ層と出力層に相当し、式 (5.3)(5.7) の順伝播と逆伝播を定義する Neural を宣言する。

NeuralNetwork.scala

```
trait Neural {
  val dim: Int
  def apply(x: Seq[Double]): Seq[Double]
  def apply(x: Seq[Double], t: Seq[Double]): Seq[Double]
}
```

2 重定義の apply メソッドで伝播を実行する。次に実装する Output クラスは、出力層を具体的に定義する。

NeuralNetwork.scala

```
class Output(val dim: Int = 1, loss: (Double,Double)=>Double = _-_) extends Neural {
  def apply(x: Seq[Double]) = x
  def apply(x: Seq[Double], t: Seq[Double]) = (x,t).zipped.map(loss)
}
```

同様に Neural を継承して Neuron クラスを実装する。Neuron クラスは、入力層ないし隠れ層の働きをする。

NeuralNetwork.scala

```
class Neuron(val dim: Int, act: Active, next: Neural, sgd: ()=>SGD) extends Neural {
  val W = Seq.fill(next.dim, dim)(sgd())
  def apply(x: Seq[Double]) = next(act.fp(z(x)))
  def apply(x: Seq[Double], t: Seq[Double]): Seq[Double] = {
    val xE = next(act.fp(z(x)), t)
    val zE = (xE, act.bp(z(x))).zipped.map(_*_)
    for((w,ze)<-W zip zE; (w,x)<-w zip x) w.update(x*ze)
    return W.transpose.map((_,zE).zipped.map(_.w*_).sum)
  }
  def z(x: Seq[Double]) = W.map((_,x).zipped.map(_.w*_).sum)
}
```

続く Offset クラスは Neuron クラスを隠蔽し、層の入力 $\boldsymbol{x}_m$ に定数 1 を追加する。定数項の役割を果たす。

NeuralNetwork.scala

```
class Offset(val dim: Int, act: Active, next: Neural, sgd: ()=>SGD) extends Neural {
  val hidden = new Neuron(dim + 1, act, next, sgd)
  def offset = hidden.W.map(_.last.w)
  def apply(x: Seq[Double]) = hidden(x:+1d)
  def apply(x: Seq[Double], t: Seq[Double]) = hidden(x:+1d, t).init
}
```

最後に SGD クラスを定義する。これは、加重 w_m^{ij} を保持すると同時に、勾配法による w_m^{ij} の更新も実装する。

NeuralNetwork.scala

```
abstract class SGD(var w: Double = math.random) {
  def update(e: Double): Unit
}
```

下記の PlainSGD クラスは、式 (5.6) に示した通りの基礎的な勾配法を実装する。学習率 η は引数 e に渡す。

NeuralNetwork.scala

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

誤差逆伝播法の実装は以上である。活性化関数 f_{sigm} と 2 乗誤差 E_{sq} の 3 層パーセプトロンの実装例を示す。

NeuralNetwork.scala

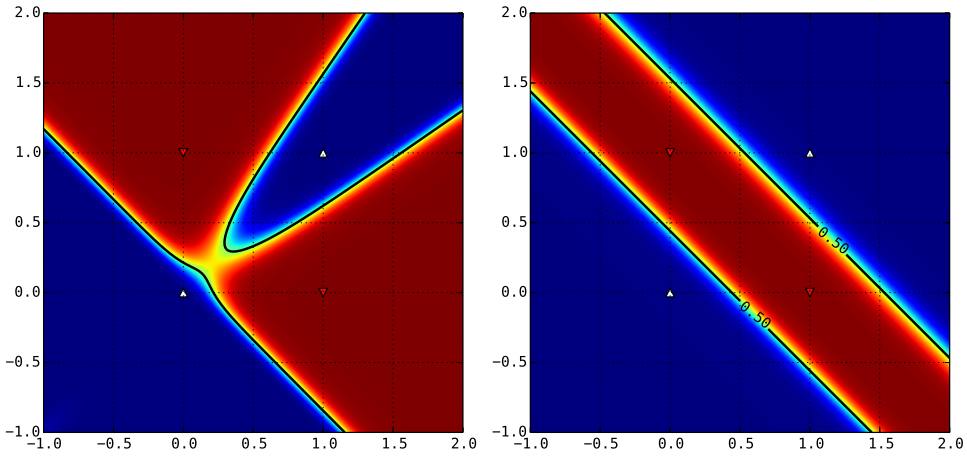
```
val model3 = new Output(1, _-_)
val model2 = new Offset(3, new Sigmoid, model3, ()=>new PlainSGD)
val model1 = new Offset(2, new Sigmoid, model2, ()=>new PlainSGD)
```

境界線が超平面では表現不可能な**非線型分離**の分類問題の例として、排他的論理和の学習を実験してみよう。

NeuralNetwork.scala

```
for(n<-1 to 1000000; x<-0 to 1; y<-0 to 1) model1(Seq(x,y), Seq(x^y))
```

Fig. 5.4(a) に定数項なしの、(b) に定数項を含む場合の学習結果を示す。定数項の有無で境界線が変化する。



(a) Neuron + Neuron.　　(b) Offset + Offset.

Fig. 5.4: exclusive OR learned by a three-layer perceptron.

同じ境界線でも、加重の初期値を変えて何度も試行すれば、収束に要する時間が変化する様子を観察できる。
余談だが、Fig. 5.2(b) に示した論理和の境界線を再現するには、線型分類器を下記の通りに構築し学習する。

NeuralNetwork.scala

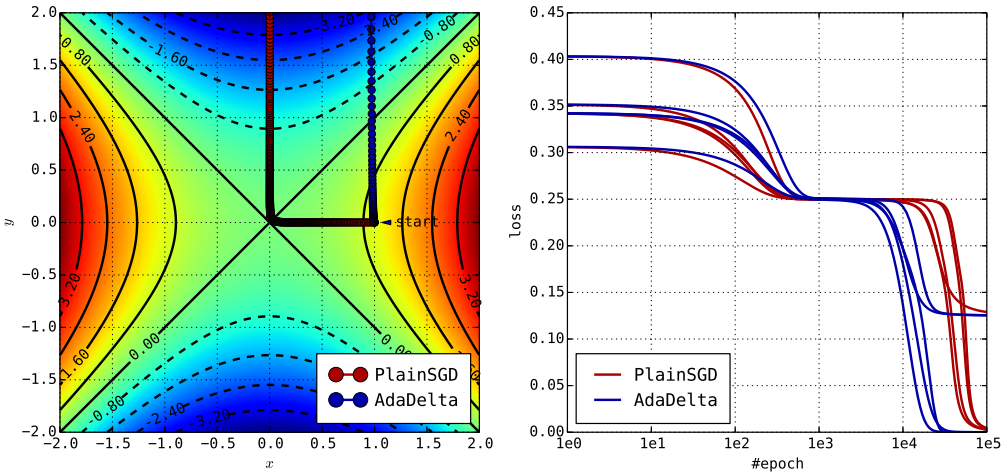
```
val model = new Offset(2, new Sigmoid, new Output, ()=>new PlainSGD)
for(n<-1 to 1000000; x<-0 to 1; y<-0 to 1) model(Seq(x,y), Seq(x|y))
```

5.2 鞍点と学習率

式 (5.10) に与える関数 E に対し、原点 $\boldsymbol{O}$ は勾配が $\nabla E = \mathbf{0}$ であるが、極小点ではない。これを**鞍点**と呼ぶ。

$$\Delta E = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y = 2x\Delta x - 2y\Delta y \ \text{ where } E(x,y) = x^2 - y^2. \tag{5.10}$$

Fig. 5.5(a) に示す通り、鞍点の近傍では式 (5.6) の勾配法は停滞し、運悪く鞍点に嵌れば、そこで収束する。Fig. 5.5(b) は、5 個の初期値で排他的論理和を学習した際の誤差 E_{sq} の推移だが、学習の停滞が観察できる。



(a) saddle point avoidance mechanism.　(b) loss diminution through training.

Fig. 5.5: comparison of `PlainSGD` and `AdaDelta`.

対策として、最適解の近傍で学習率を小さく、鞍点の近傍では大きく設定する**適応的勾配法**が効果的である。式 (5.11) に示す *AdaGrad* は、学習率を勾配 ∇E_{mt}^{ij} の期待値に反比例させつつ、時刻 t に伴って減衰させる。

$$\Delta w_{mt}^{ij} = -\frac{\eta}{t\sqrt{\mathbf{E}\Big[\,(\nabla E_m^{ij})^2\,\Big]_t}} \ \text{ where } \mathbf{E}\Big[\,(\nabla E_m^{ij})^2\,\Big]_t = \frac{1}{t}\sum_{\tau=0}^{t}(\nabla E_{mt}^{ij})^2,\ \mathbf{E}\Big[\,(\nabla E_m^{ij})^2\,\Big]_0 = \varepsilon. \tag{5.11}$$

式 (5.12) に示す *AdaDelta* は、直近の勾配を重視する。加重 w_m^{ij} と勾配 ∇E_m^{ij} の**スケール変換**の効果も持つ。

$$\Delta w_{mt}^{ij} = -\frac{\sqrt{\mathbf{E}\Big[\,(\Delta w_m^{ij})^2\,\Big]_t + \varepsilon}}{\sqrt{\mathbf{E}\Big[\,(\nabla E_m^{ij})^2\,\Big]_t + \varepsilon}}\nabla E_{mt}^{ij} \ \text{ where } \mathbf{E}[\,x\,]_t = \rho\mathbf{E}[\,x\,]_{t-1} + (1-\rho)x_t,\ \mathbf{E}[\,x\,]_0 = 0. \tag{5.12}$$

下記の `AdaDelta` クラスに実装する。引数 `r` は係数 ρ を表す。また、引数 `e` は**ゼロ除算**を防ぐ係数 ε である。

`StochasticGradientDescent.scala`

```
class AdaDelta(r: Double = 0.95, e: Double = 1e-8) extends SGD {
  var eW,eE = 0.0
  def update(E: Double) = {
    this.eE = r*eE + (1-r) * math.pow(1*E, 2)
    val n = math.sqrt(eW+e) / math.sqrt(eE+e)
    this.eW = r*eW + (1-r) * math.pow(n*E, 2)
    this.w -= n * E
  }
}
```

`PlainSGD` クラスで Fig. 5.5(a) の軌跡を描くには 419 回の更新を要したが、`AdaDelta` クラスは 66 回である。

5.3 多クラス分類

出力 $y \in \{1, ..., K\}$ の分類器の活性化関数は、関数 f_{sigm} を K クラスに拡張した**ソフトマックス関数**を使う。

$$y \sim q(y) = f_{\text{smax}}(\boldsymbol{z}) = \frac{f_{\exp}(\boldsymbol{z})}{\|f_{\exp}(\boldsymbol{z})\|} \text{ where } f_{\exp}(\boldsymbol{z}) = {}^t\left(e^{z_1} \cdots e^{z_K}\right) . \tag{5.13}$$

直感的には、最適な事後分布 $q(y)$ は正解 $p(t)$ との差を表す Kullback–Leibler 情報量 $D(p\|q)$ を最小化する。Kullback–Leibler 情報量は非負で、式 (5.14) に示す**ギブスの不等式**より $q=p$ に限り $D(p\|q)=0$ が成立する。

$$D(p\|q) = \int_K p(y) \log \frac{p(y)}{q(y)} dy \geq \int_K p(y) \left(1 - \frac{q(y)}{p(y)}\right) dy = 0 \ \leftarrow \log x \leq x - 1. \tag{5.14}$$

通常は $D(p\|q)$ の上限を与える式 (5.5) の交差エントロピー E_{CE} を通じて、間接的に $D(p\|q)$ を最小化する。理由は、出力層の直前の勾配 $\nabla E_{\text{CE}}(\boldsymbol{z})$ が式 (5.15) で容易に求まり、式 (5.5) で $H(p)$ が定数である点にある。

$$\frac{\partial E_{\text{CE}}}{\partial z_k} = -\frac{\partial}{\partial z_k} \sum_{i=1}^{K} t_i \left(\log e^{z_i} - \log \sum_{j=1}^{K} e^{z_j}\right) = -t_k + \sum_{i=1}^{K} t_i y_k = -t_k + y_k. \tag{5.15}$$

下記の `Softmax` クラスに式 (5.13) を実装する。出力層の直前での利用を前提に、逆伝播は恒等関数にした。

`SoftmaxFunction.scala`

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

`Softmax` クラスは、出力層の直前での利用に限定される点を除き、他の活性化関数と同じ要領で利用できる。

`SoftmaxFunction.scala`

```
val model3 = new Output(4, _-_)
val model2 = new Offset(3, new Softmax, model3, ()=>new PlainSGD)
val model1 = new Offset(2, new Sigmoid, model2, ()=>new PlainSGD)
```

Fig. 5.6 は、xy 軸上の 4 点を標本に与えて、3 層パーセプトロンで国際信号旗の *zulu* を学習した結果である。

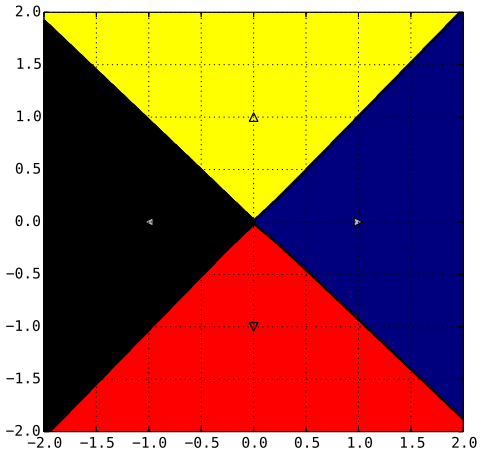


Fig. 5.6: maritime signal flag *zulu* learned by a three-layer perceptron.

原理上は、第 5.1 節と同様に 2 乗誤差 E_{sq} で学習できるが、勾配 ∇E_{sq} が緩慢なため、収束に時間がかかる。

第6章　サポートベクターマシン

サポートベクターマシンとは、Fig. 6.1 に示す**判別境界**を学習し、標本 $\{\boldsymbol{x}_i\}$ を正負に分割する分類器である。両側の集団からの最短距離 d が最大になる直線を学習することで、未知の $\boldsymbol{x}$ を誤分類する可能性を抑制する。

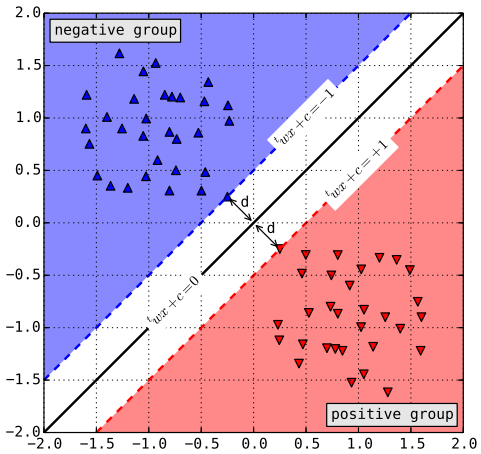


Fig. 6.1: a support vector machine.

判別境界 $\boldsymbol{w}\cdot\boldsymbol{x}+c=0$ の学習は**制約付き最大化問題**であり、標本 $\{\boldsymbol{x}_i, t_i\}$ は条件式 (6.1) を満たす必要がある。

$$t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)\geq 1 \ \text{ where } t_i=\begin{cases}+1 & \text{if } \boldsymbol{x}_i\in\text{positive group}\\ -1 & \text{if } \boldsymbol{x}_i\in\text{negative group}\end{cases} \tag{6.1}$$

ただし、t_i は点 $\boldsymbol{x}_i$ が帰属する集団を示す変数である。集団と判別境界との距離 d は、式 (6.2) で計算できる。

$$d(\{\boldsymbol{x}_i\})=\min\frac{|\boldsymbol{w}\cdot\boldsymbol{x}_i+c|}{\|\boldsymbol{w}\|}=\frac{1}{\|\boldsymbol{w}\|}. \tag{6.2}$$

現実には、正負の境界が曖昧な場合には、式 (6.1) の遵守が困難なため、**ソフトマージン**による緩和を図る。即ち、制約条件を式 (6.3) に変更し、若干の誤分類を見逃す代わりに、誤分類点 $\boldsymbol{x}_i$ にヒンジ損失 ξ_i を課す。

$$t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)\geq 1-\xi_i \ \text{ where } \xi_i=\begin{cases}0 & \text{if } t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)>1\\ |t_i-(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)| & \text{if } t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)\leq 1.\end{cases} \tag{6.3}$$

誤分類の抑制と距離 d の最大化は、式 (6.4) の最小化で達成する。ここで、$\|\boldsymbol{w}\|^2$ は L2 正則化の役割を担う。

$$f(\{\boldsymbol{x}_i, t_i\}, \boldsymbol{w}, c)=C\sum_{i=1}^{N}\xi_i+\frac{1}{2}\|\boldsymbol{w}\|^2 \text{ where } C>0. \tag{6.4}$$

誤分類点の集合 $E(\{\boldsymbol{x}_i\})$ に対し、式 (6.5) が成立する。故に、定数 C は誤分類率の限度を間接的に決定する。

$$\sum_{i=1}^{N}\xi_i>|E(\{\boldsymbol{x}_i\})| \leftarrow \forall\boldsymbol{x}_i\in E(\{\boldsymbol{x}_i\}),\ \xi_i>1. \tag{6.5}$$

6.1 凸二次計画問題

式 (6.4) の最小化は、未定乗数法と Karush–Kuhn–Tucker 条件の適用により、**凸二次計画問題**で達成できる。

$$L(\boldsymbol{w},c,\xi,\lambda,\mu)=\frac{1}{2}\|\boldsymbol{w}\|^2+C\sum_{i=1}^{N}\xi_i-\sum_{i=1}^{N}\lambda_i\{t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+c)-1\}-\sum_{i=1}^{N}\mu_i\xi_i. \tag{6.6}$$

λ_i と μ_i は座標 $\boldsymbol{x}_i$ に関する未定乗数である。係数 $\boldsymbol{w}$ と c で偏微分して、L が最小になる条件式 (6.7) を得る。

$$\tilde{L}(\lambda)=\min_{\boldsymbol{w},c}L(\boldsymbol{w},c,\lambda)=\sum_{i=1}^{N}\lambda_i\left\{1-\frac{1}{2}\sum_{j=1}^{N}\lambda_j t_i t_j(\boldsymbol{x}_i\cdot\boldsymbol{x}_j)\right\}\ \text{where}\ \begin{cases}0=\displaystyle\sum_{i=1}^{N}\lambda_i t_i\\ \lambda_i=C-\mu_i\\ \boldsymbol{w}=\displaystyle\sum_{i=1}^{N}\lambda_i t_i\boldsymbol{x}_i\end{cases}. \tag{6.7}$$

式 (6.3) は不等式ゆえ、Karush–Kuhn–Tucker 条件より、式 (6.7) が最適解を与える場合、式 (6.8) を満たす。誤分類点の存在を無視すると、式 (6.7)(6.8) より、判別境界から距離 d の点のみが係数 $\boldsymbol{w}$ の決定に寄与する。

$$\lambda_i\left\{t_i\left(\sum_{j=1}^{N}\lambda_j t_j\boldsymbol{x}_j\cdot\boldsymbol{x}_i+c\right)+\xi_i-1\right\}=0\ \text{where}\ \begin{cases}\lambda_i\geq 0\\ \mu_i\geq 0\\ \mu_i\xi_i=0\end{cases}. \tag{6.8}$$

これを**サポートベクトル**と呼ぶ。より遠方の点は、$\lambda_i=0$ を満たす必要から、係数 $\boldsymbol{w}$ の決定に無関係である。以後、双対問題 $\tilde{L}(\lambda)$ を最大化する。今回は、実装が容易で計算が高速な**逐次最小問題最適化法**を利用する。

$$t_i\delta_i+t_j\delta_j=0\ \text{where}\ \begin{cases}\delta_i=\hat{\lambda}_i-\lambda_i\\ \delta_j=\hat{\lambda}_j-\lambda_j\end{cases}\leftarrow 0=\sum_{i=1}^{N}\lambda_i t_i. \tag{6.9}$$

これは、式 (6.8) を破る点 $\boldsymbol{x}_i$ が存在する限り、任意の点 $\boldsymbol{x}_j$ を選び、式 (6.9) を満たす局所的な最適化を施す。式 (6.7) の $\tilde{L}(\lambda)$ から、1 回の最適化による λ_i,λ_j の増分 δ_i,δ_j を含む部分式を抜き出すと、式 (6.10) を得る。

$$d\tilde{L}(\delta_i,\delta_j)=\delta_i+\delta_j-\frac{1}{2}|\delta_i t_i\boldsymbol{x}_i+\delta_j t_j\boldsymbol{x}_j|^2-\sum_{n=1}^{N}\lambda_n t_n\boldsymbol{x}_n\cdot(\delta_i t_i\boldsymbol{x}_i+\delta_j t_j\boldsymbol{x}_j). \tag{6.10}$$

式 (6.10) に式 (6.9) の制約を加えると、式 (6.11) を得る。なお、式 (6.11) は変数 t_i の符号に拘らず成立する。

$$d\tilde{L}(\delta_i)=t_i(t_i-t_j)\delta_i-\frac{1}{2}\delta_i^2|\boldsymbol{x}_i-\boldsymbol{x}_j|^2-t_i\delta_i\sum_{n=1}^{N}\lambda_n t_n\boldsymbol{x}_n\cdot(\boldsymbol{x}_i-\boldsymbol{x}_j). \tag{6.11}$$

双対問題 $\tilde{L}(\lambda)$ の最大化は、$d\tilde{L}(\delta_i,\delta_j)$ の極大点の探索と同義である。δ_i での偏微分により、式 (6.12) を得る。なお、更新後の値 $\hat{\lambda}_i$ と $\hat{\lambda}_j$ が式 (6.8) を逸脱する場合は、**クリッピング**により強制的に区間 $[0,C]$ に収める。

$$\delta_i=-\frac{t_i}{|\boldsymbol{x}_i-\boldsymbol{x}_j|^2}\left\{\sum_{n=1}^{N}\lambda_n t_n\boldsymbol{x}_n\cdot(\boldsymbol{x}_i-\boldsymbol{x}_j)-t_i+t_j\right\}. \tag{6.12}$$

座標 $\boldsymbol{x}_i$ に対し、条件式 (6.8) を確認する際は、係数 $\boldsymbol{w}$ と c の値が必要になるが、$\boldsymbol{w}$ の値は式 (6.7) で求まる。定数項 c の値は、$t_i(\boldsymbol{w}\cdot\boldsymbol{x}_i)$ を最小化する座標 $\boldsymbol{x}_i$ がサポートベクトルとなる点に着目し、式 (6.13) で求める。

$$c=-\frac{1}{2}\left\{\min_{i|t_i=+1}\sum_{j=1}^{N}\lambda_j t_j\boldsymbol{x}_i\cdot\boldsymbol{x}_j+\max_{j|t_j=-1}\sum_{i=1}^{N}\lambda_i t_i\boldsymbol{x}_j\cdot\boldsymbol{x}_j\right\}. \tag{6.13}$$

下記の `SVM` クラスはサポートベクターマシンを実装する。標本 $\{\boldsymbol{x}_i\}$ は引数 `X` に渡す。引数 `k` は内積を表す。

SupportVectorMachine.scala

```
class SVM(X: Seq[(Seq[Double],Int)], C: Double, k: (Seq[Double],Seq[Double])=>Double) {
  val data = X.map(Data.tupled)
  var c = 0.0
```

SVM クラス内に Data クラスを実装する。これは、標本の i 番目の点 $\boldsymbol{x}_i$ と正解 t_i と未定乗数 λ_i を記憶する。

SupportVectorMachine.scala

```
  case class Data(x: Seq[Double], t: Int) {
    var l = 0.0
  }
```

判別境界を表す係数 $\boldsymbol{w}$ は変数に記憶せず、内積 $\boldsymbol{w}\cdot\boldsymbol{x}$ の形で実装する。これは第 6.2 節に向けた布石である。

SupportVectorMachine.scala

```
  def wx(x: Data) = data.map(d => d.l * d.t * k(x.x,d.x)).sum
```

kkt メソッドは、式 (6.8) の Karush–Kuhn–Tucker 条件を検査する。実数の比較では**丸め誤差**に注意を払う。

SupportVectorMachine.scala

```
  def kkt(x: Data) = (x.t * this(x.x) - 1) match {
    case e if e < -1e-10 => x.l >= C
    case e if e > +1e-10 => x.l <= 0
    case _ => true
  }
```

clip メソッドは、式 (6.12) の最適化を行う際に、未定定数の差分 δ_i と δ_j の閉区間 $[0, C]$ からの逸脱を防ぐ。

SupportVectorMachine.scala

```
  def clip(xi: Data, xj: Data, d: Double): Double = {
    if(d < lower(xi, xj)) return lower(xi, xj)
    if(d > upper(xi, xj)) return upper(xi, xj)
    return if(d.isNaN) 0 else d
  }
```

lower メソッドは、差分 δ_i や δ_j の下限を計算する。下限は式 (6.9) の δ_i の定義と式 (6.7)(6.8) から導ける。

SupportVectorMachine.scala

```
  def lower(xi: Data, xj: Data) = xi.t * xj.t match {
    case  1 => math.max(-xi.l, +xj.l - C)
    case -1 => math.max(-xi.l, -xj.l)
  }
```

同様に、差分 δ_i や δ_j の上限も式 (6.9) の δ_i の定義と式 (6.7)(6.8) から導出し、upper メソッドに実装する。

SupportVectorMachine.scala

```
  def upper(xi: Data, xj: Data) = xi.t * xj.t match {
    case  1 => math.min(+xj.l, -xi.l + C)
    case -1 => math.min(-xi.l, -xj.l) + C
  }
```

次に実装する pos と neg は、正負のサポートベクトルの内積 $\boldsymbol{w}\cdot\boldsymbol{x}$ を求める。式 (6.13) を計算する際に使う。

SupportVectorMachine.scala

```
  def pos = data.filter(_.t == +1).map(wx(_)).min
  def neg = data.filter(_.t == -1).map(wx(_)).max
```

下記は、逐次最小問題最適化法の反復処理である。式 (6.8) を破る点 $\boldsymbol{x}_i$ を探し、式 (6.12) に従って更新する。

SupportVectorMachine.scala

```
  while(data.count(!kkt(_)) >= 2) {
    val a = data.find(!kkt(_)).get
    val b = data(util.Random.nextInt(X.size))
    val sub = wx(Data((a.x,b.x).zipped.map(_-_), 0))
    val den = k(a.x,a.x) - 2*k(a.x,b.x) + k(b.x,b.x)
    val del = clip(a, b, -a.t * (sub-a.t+b.t) / den)
    a.l += del
    b.l -= del * a.t * b.t
    this.c = -0.5 * (pos+neg)
  }
```

下記の apply メソッドは、未知の点 $\boldsymbol{x}$ に対して $\boldsymbol{w}\cdot\boldsymbol{x}+c$ を計算する。返り値の正負は、帰属する集団を表す。

SupportVectorMachine.scala

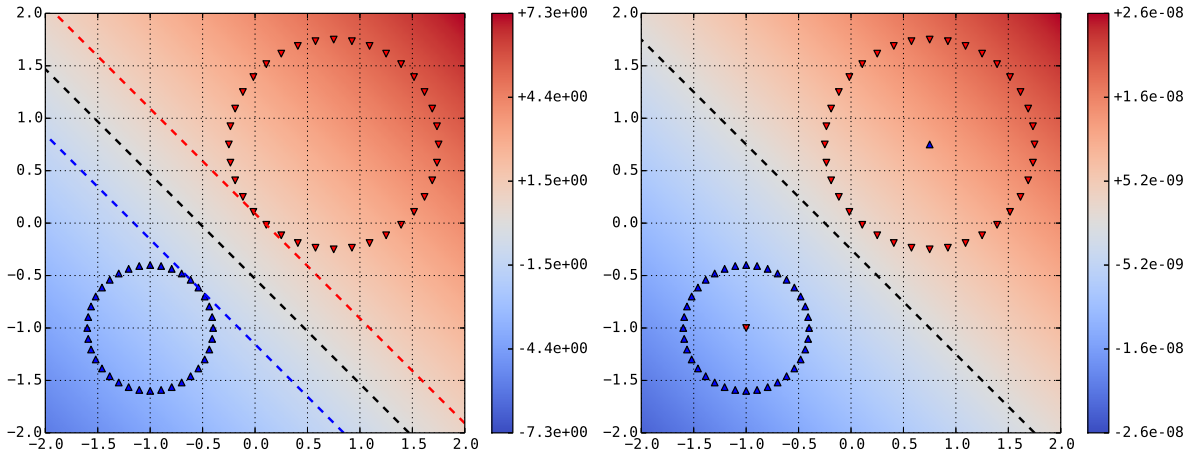
```
  def apply(x: Seq[Double]) = wx(Data(x, 0)) + c
}
```

完成した SVM クラスを使う際は、標本 $\boldsymbol{x}_i$ に加え、ソフトマージンの定数 C と内積の定義を引数に指定する。

SupportVectorMachine.scala

```
val svm = new SVM(data, 1e-10, (a, b) => (a, b).zipped.map(_*_).sum)
```

Fig. 6.2 は、線型分離可能な標本を SVM クラスで学習した結果である。ただし (b) は 2 個の誤分類点を含む。黒の点線は判別境界を表し、赤と青の点線は $\boldsymbol{w}\cdot\boldsymbol{x}+c=\pm 1$ を表す。赤と青の濃淡は $\boldsymbol{w}\cdot\boldsymbol{x}+c$ の相対値を表す。



(a) sample data separable by a line.

(b) sample data with outlier points.

Fig. 6.2: decision surface learned by a linear SVM.

(a) は、両集団を直線で完璧に分離できる場合、判別境界がサポートベクトルの中間に位置する様子を表す。(b) は、判別境界が正の領域にある負の誤分類点に誘引され、かつサポートベクトルが消滅する様子を表す。

6.2 特徴空間の変換

第 6.2 節では、サポートベクターマシン等の**線型分類器**を非線型分離な問題に対応させる**カーネル法**を学ぶ。根底には、標本を高次元空間に写像することで、像の分布を疎にした上で、線型分離を達成する着想がある。

$$\Phi : \boldsymbol{x} \mapsto \Phi_{\boldsymbol{x}} \quad \text{e.g.} \quad \begin{pmatrix} x \\ y \end{pmatrix} \mapsto z = \frac{1}{2\pi} \exp\left(-\frac{x^2 + y^2}{2} \right). \tag{6.14}$$

Fig. 6.3 は、同心円に並ぶ正負の集団を式 (6.14) で z 軸上に投影し、z 軸に垂直な分離平面を得る例である。

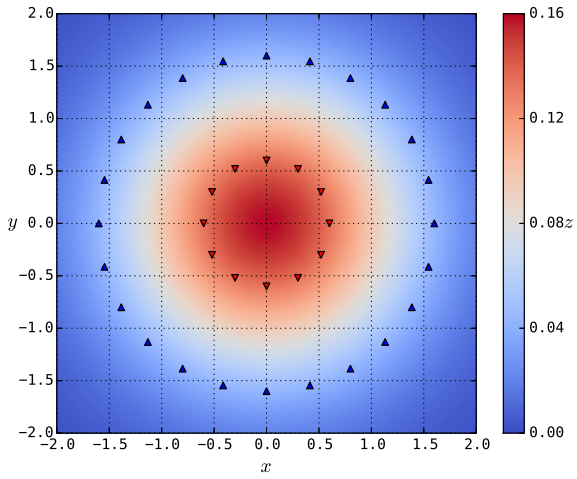


Fig. 6.3: conversion of linearly inseparable problem to separable.

式 (6.1–6.13) の座標 $\boldsymbol{x}$ を $\Phi_{\boldsymbol{x}}$ に変更し、第 6.2 節の議論に流用する。例えば、係数 $\boldsymbol{w}$ は式 (6.15) で求める。基本的に、像 $\Phi_{\boldsymbol{x}}$ の次元 D が高ければ分離の機会は増えるが、式 (6.1–6.13) の計算は $\mathcal{O}(D)$ の時間を要する。

$$\boldsymbol{w} = \sum_{i=1}^{N} \lambda_i t_i \Phi_{\boldsymbol{x}_i}. \tag{6.15}$$

ここで、式 (6.8)(6.12)(6.13) を注意深く観察すると、内積 $\Phi_{\boldsymbol{x}_i} \cdot \Phi_{\boldsymbol{x}_j}$ が求まれば $\Phi_{\boldsymbol{x}}$ の値は不要な点に気付く。実は、像の計算を経ず内積を計算する手段が存在する。まず、式 (6.16) に示す 2 変数の**対称関数**を定義する。

$$k : \mathbb{M} \times \mathbb{M} \to \mathbb{R} \quad \text{where } k(\boldsymbol{x}_i, \boldsymbol{x}_j) = k(\boldsymbol{x}_j, \boldsymbol{x}_i). \tag{6.16}$$

記号 $\mathbb{M}$ は**可測空間**を指す。関数 k は、式 (6.17) に示す**正定値性**を満たす場合、**正定値カーネル**と呼ばれる。

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \ {}^t\boldsymbol{x} \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix} \boldsymbol{x} > 0. \tag{6.17}$$

式 (6.17) の関数 k に対し、式 (6.18) の像 $\Phi_{\boldsymbol{x}_i}$ の線型結合からなる空間 H_k を**再生核ヒルベルト空間**と呼ぶ。

$$H_k = \left\{ f(\boldsymbol{x}) = \sum_{n=1}^{N} a_n \Phi_{\boldsymbol{x}_n} = \sum_{n=1}^{N} a_n k(\boldsymbol{x}, \boldsymbol{x}_n) \right\} \text{ where } \Phi : \boldsymbol{x} \mapsto \Phi_{\boldsymbol{x}_n} = k(\boldsymbol{x}, \boldsymbol{x}_n). \tag{6.18}$$

空間 H_k の元 f と g に対し、内積 $\langle f, g \rangle$ は非退化かつ正定値な**対称双線型形式**であり、式 (6.19) で定義する。

$$\langle f, g \rangle = \sum_{i=1}^{N} \sum_{j=1}^{N} a_i b_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \quad \text{where} \quad \left\{ \begin{aligned} f(\boldsymbol{x}) &= \sum_{i=1}^{N} a_i k(\boldsymbol{x}, \boldsymbol{x}_i) \\ g(\boldsymbol{x}) &= \sum_{j=1}^{N} b_j k(\boldsymbol{x}, \boldsymbol{x}_j) \end{aligned} \right\} \ a_i, b_j \in \mathbb{R}. \tag{6.19}$$

余談ながら、内積と距離を備え、かつ**完備**な空間を**ヒルベルト空間**と呼び、式 (6.20) の性質を**再生性**と呼ぶ。

$$\langle f, k(\boldsymbol{x}, \boldsymbol{x}_j) \rangle = \sum_{i=1}^{N} a_i k(\boldsymbol{x}_i, \boldsymbol{x}_j) = f(\boldsymbol{x}_j). \tag{6.20}$$

式 (6.20) の性質は、像 $\Phi_{\boldsymbol{x}_i}$ や $\Phi_{\boldsymbol{x}_j}$ の明示的な計算を経ずに、内積 $\Phi_{\boldsymbol{x}_i} \cdot \Phi_{\boldsymbol{x}_j}$ の値が求まる可能性を示唆する。この技法は**カーネルトリック**と通称される。関数 k の定義次第では、無限次元の空間への変換も可能である。

$$\Phi_{\boldsymbol{x}_i} \cdot \Phi_{\boldsymbol{x}_j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \begin{cases} \displaystyle\sum_{d=1}^{D} x_{id} x_{jd} & \cdots \text{ linear kernel} \\ \displaystyle\exp\left(-\frac{|\boldsymbol{x}_i - \boldsymbol{x}_j|^2}{2\sigma^2} \right) & \cdots \text{ Gaussian kernel} \end{cases} \tag{6.21}$$

式 (6.21) のうち**線型カーネル**は単なる恒等変換に過ぎないが、**ガウシアンカーネル**には興味深い特性がある。

$$\exp \frac{\boldsymbol{x}_i \cdot \boldsymbol{x}_j}{\sigma^2} = \exp \sum_{d=1}^{D} \frac{x_{id} x_{jd}}{\sigma^2} = \prod_{d=1}^{D} \exp \frac{x_{id} x_{jd}}{\sigma^2} = \prod_{d=1}^{D} \sum_{n=0}^{\infty} \frac{1}{\sqrt{n!}} \left(\frac{x_{id}}{\sigma} \right)^n \frac{1}{\sqrt{n!}} \left(\frac{x_{jd}}{\sigma} \right)^n. \tag{6.22}$$

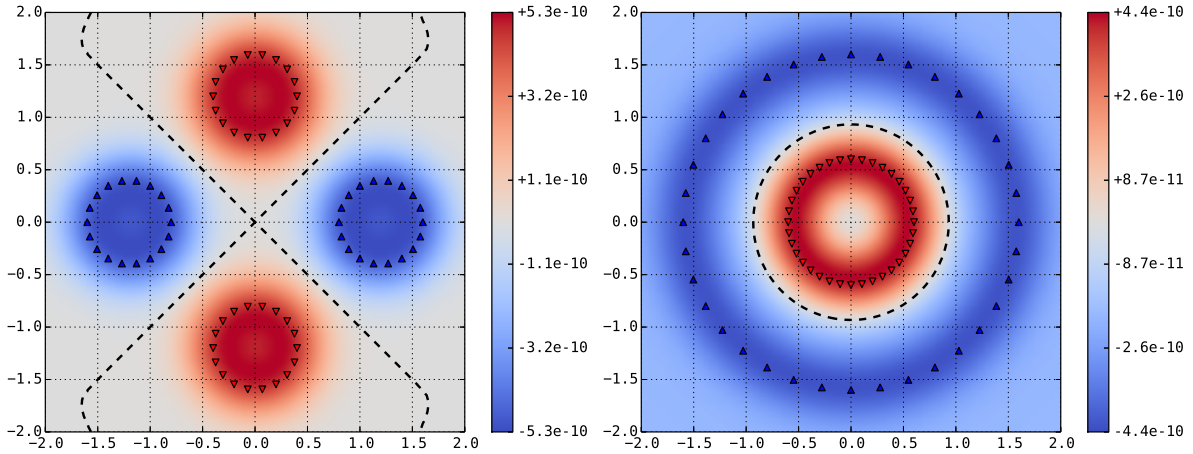
式 (6.22) は、式 (6.23) のような無限次元のベクトルの内積と同義であり、事実上、無限次元への写像を表す。

$$\sum_{n=0}^{\infty} \frac{1}{\sqrt{n!}} \left(\frac{x_{id}}{\sigma} \right)^n \frac{1}{\sqrt{n!}} \left(\frac{x_{jd}}{\sigma} \right)^n = \begin{pmatrix} \frac{1}{\sqrt{0!}} \left(\frac{x_{id}}{\sigma} \right)^0 \\ \vdots \\ \frac{1}{\sqrt{n!}} \left(\frac{x_{id}}{\sigma} \right)^n \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{0!}} \left(\frac{x_{jd}}{\sigma} \right)^0 \\ \vdots \\ \frac{1}{\sqrt{n!}} \left(\frac{x_{jd}}{\sigma} \right)^n \\ \vdots \end{pmatrix} \tag{6.23}$$

他の著名なカーネルの例では、式 (6.24) の**シグモイドカーネル**は 3 層パーセプトロンに似た挙動で知られる。

$$\operatorname{sign}(\boldsymbol{w} \cdot \Phi_{\boldsymbol{x}} + c) = \operatorname{sign}\left(\sum_{i=1}^{N} \lambda_i t_i k_s(\boldsymbol{x}, \boldsymbol{x}_i) + c \right) \quad \text{where } k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh\left(c \sum_{d=1}^{D} x_{id} x_{jd} + \theta \right). \tag{6.24}$$

Fig. 6.4 は、`SVM` クラスの引数 `k` にガウシアンカーネルを与えて、非線型分離な標本を学習した結果である。



(a) diamond-shaped samples.　　(b) two concentric circles.

Fig. 6.4: decision surface learned by a Gaussian SVM.

黒の点線は判別境界を表し、赤と青の濃淡は $\boldsymbol{w} \cdot \boldsymbol{x} + c$ の相対値を表す。カーネル法の利点がよく理解できる。