Scala で実装する機械学習入門

Let's try machine learning algorithms by Scala

無線部開発班

2015 年 9 月 15 日

1 はじめに

本稿では、ニューラルネットワークやサポートベクターマシンなど、代表的な機械学習アルゴリズムを導出を交えつつ Scala で実装する。機械学習はこれが全てではないが、感覚を掴む上では有用となろう。

2 最近傍探索

k 近傍法は、あるデータの近傍にある他の k 個のデータに多数決を取ってもらい分類する手法である。Fig. 1, 2 は 6 個の赤い正のサンプル点と 6 個の青い負のサンプル点に対して k 近傍法を行い、空間を正と負の領域に区分した例である。Fig. 1 は $k = 2$ とした場合で、Fig. 2 は $k = 3$ とした場合である。

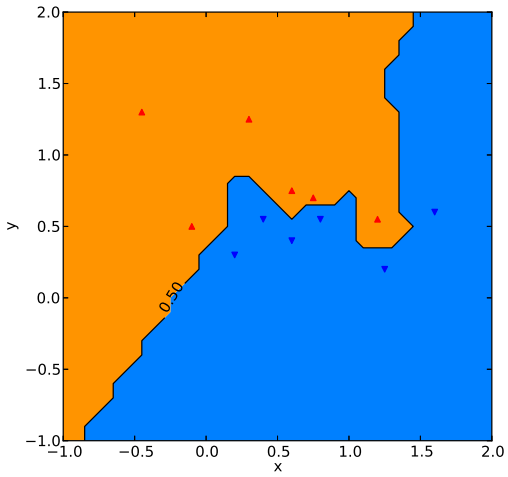

Fig. 1 $k = 2$ nearest neighbors.

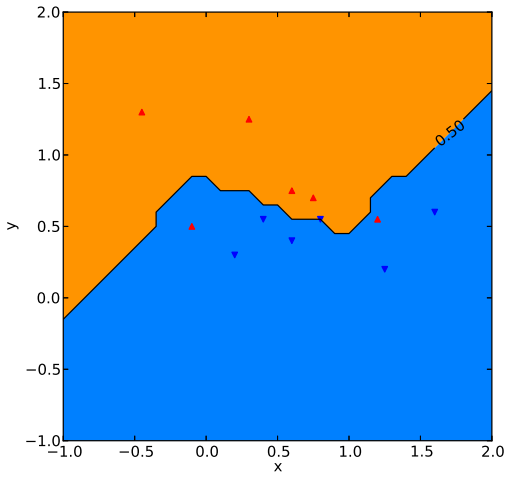

Fig. 2 $k = 3$ nearest neighbors.

k の選び方によって結果が変わるのがわかる。実は、k 近傍法を使うにあたってはこの k の適切な値の設定方法が問題になるのだが、本稿では扱わない。また、距離をどのように定義するかも重要な課題である。典型的な例としてはユークリッド距離やマンハッタン距離があるが、他にも距離の公理を満たす関数であれば利用できる。k 近傍法はクラス分類手法としては最も単純なアルゴリズムであり、事前の学習が不要で、分類時に初めて計算を行う点が特徴的である。こうした特徴から**遅延学習**とも呼ばれる。

```
case class Data(point: Seq[Double], answer: Any = 0)
```

`Data` クラスは 1 件のサンプルデータを表現するケースクラスで、座標内の位置と正負の正解ラベルを持つ。例えば、Fig. 1, 2 に示した例のサンプルデータは以下のように定義してある。1 が正のラベルで 0 が負のラベルである。今回は Python の matplotlib を用いて図を描く都合でラベルを数値としたが `Data` クラスでの宣言は `Any` 型なのでどんなオブジェクトでもラベルとして利用できる。

```
val samples = Seq(
  Data(Seq(-0.10, +0.50), 1), Data(Seq(-0.45, +1.30), 1),
  Data(Seq(+0.60, +0.75), 1), Data(Seq(+0.30, +1.25), 1),
  Data(Seq(+0.75, +0.70), 1), Data(Seq(+1.20, +0.55), 1),
  Data(Seq(+0.20, +0.30), 0), Data(Seq(+1.60, +0.60), 0),
  Data(Seq(+0.40, +0.55), 0), Data(Seq(+0.60, +0.40), 0),
  Data(Seq(+0.80, +0.55), 0), Data(Seq(+1.25, +0.20), 0))
```

学習用の訓練データも準備できたので、さっそく k 近傍法の本体を実装してみよう。

```
class KNN(k: Int, samples: Seq[Data], distance: (Data, Data) => Double) {
  def predict(target: Data): Any = {
    val knears = samples.sortWith((t1, t2) => {
      val d1 = distance(t1, target)
      val d2 = distance(t2, target)
      d1 < d2
    }).take(k).map(_.answer)
    val answers = knears.toSet
    val verdict = answers.map(ans => ans -> knears.count(_ == ans))
    verdict.maxBy(tuple => tuple._2)._1
  }
}
```

KNN クラスが k 近傍法の本体である。以下のように k の値と訓練データと距離関数を与えて使用する。

```
val euclid = (a: Data, b: Data) => Math.sqrt(
  Math.pow(a.point(0) - b.point(0), 2) +
  Math.pow(a.point(1) - b.point(1), 2)
)
val knn = new KNN(3, samples, euclid) // k = 3
println(knn.predict(Data(Seq(0.0, 1.0))))
println(knn.predict(Data(Seq(1.0, 0.0))))
```

predict メソッドを呼び出すと、全ての訓練データに対して距離を求め、最も近い k 個の訓練データを取り出してラベルの多数決を行う。なお、実際は距離に応じた発言力の重み付けが必要になる可能性がある。例えば、Fig. 3 に示すようにある集団が局所的に囲い込まれている状況も考えられるためである。

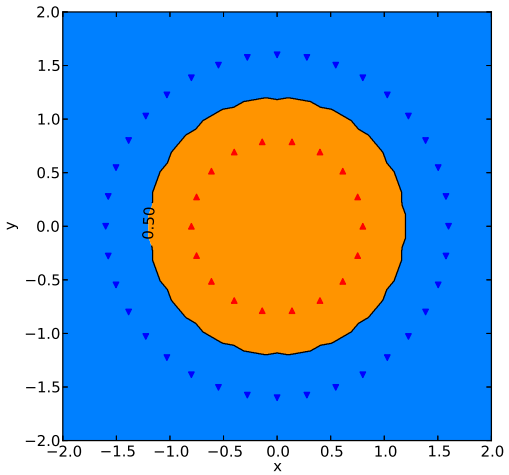

Fig. 3 concentric circular neighbors.

また、分類のたびに全てのサンプルとの距離を求めるのは計算量の点で不利である。少なくとも用途に注意を要するクラス分類手法と言えよう。今回は平面上の座標を扱ったが、分類したいデータは画像や音声や文字列など様々なデータが考えられる。これらを扱う際には何らかの方法でベクトルへの変換を行うことになるが、変換後のベクトルが定義されるベクトル空間を**特徴空間**ないし**素性空間**と呼ぶ。

3 ニューラルネットワーク

ニューラルネットワークは脳神経の仕組みをエミュレートした数学モデルで、最も歴史の長い機械学習アルゴリズムのひとつである。今回はその中でも、入力から出力へと一方向に信号が伝搬し、途中でループしない**フィードフォワード**型のニューラルネットワークを実装する。形式的に説明すると難しく思われるが、非常に単純なアルゴリズムである。分厚い教科書を購入する前に、まずは感覚を掴もう。

3.1 単純パーセプトロン

単純パーセプトロンは最も単純なニューラルネットワークである。何のことはない。n 個の入力 x_k の各々に重み w_k を付けて合計した値に**活性化関数** f を適用するだけである (式 (1))。Fig. 4 に図示する。

$$\boldsymbol{y} = f(\boldsymbol{w} \cdot \boldsymbol{x}) = f\left(w_0 + \sum_{k=1}^{n} w_k x_k\right) \tag{1}$$

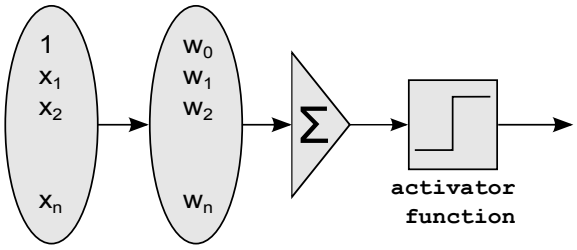

Fig. 4 single layer perceptron.

活性化関数とは、実数値の入力を受け取って、閾値以上なら `true`、未満なら `false` に分類する関数のことだと理解すれば今のところは差し支えない。パーセプトロンの入力は実数値であり、その重み付け線型和を使ってクラス分類をしたいなら、真偽値に変換しなければならない。そのために活性化関数が必要になる。グラフを描いてみればわかるが、パーセプトロンは空間を `true` と `false` の領域に分ける**決定境界**を用いて分類する機械である。例えば、論理和をパーセプトロンで再現する場合は、Fig. 5 の直線が決定境界になる。直線より上側では `true`、下側では `false` に分類される。

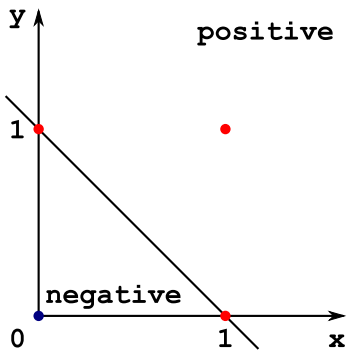

Fig. 5 decision boundary of OR operator.

直線上の点の扱いは `true` でも `false` でも構わないが、本節では便宜上 `true` としておく。決定境界を表す重みベクトル $\boldsymbol{w}$ は学習により習得する。学習については後述するとして、先に分類器を実装する。

```
class Perceptron(samples: Seq[(Seq[Double], Boolean)], dim: Int) {
  val weights = new Array[Double](dim + 1)
  def predict(in: Seq[Double]) = ((1.0 +: in) zip weights).map {
    case (i, w) => i * w
  }.sum >= 0
}
```

weights は重み $\boldsymbol{w}$ に相当する。dim は入力ベクトル $\boldsymbol{x}$ の階数である。weights の階数を dim + 1 に設定しているが、これは定数項 w_0 の分である。predict メソッドは Fig. 4 を図の通りにプログラムに起こしたものである。続いて学習の仕組みを考えてみよう。$\boldsymbol{x}_n$ の正解を $t_n = \pm 1$ として、誤差関数を

$$E_n = \begin{cases} t_n(\boldsymbol{w} \cdot \boldsymbol{x}_n) & (\boldsymbol{y}_{output} \neq \boldsymbol{y}_{answer}) \\ 0 & (\boldsymbol{y}_{output} = \boldsymbol{y}_{answer}) \end{cases} \tag{2}$$

と定義する。学習の目的は誤差 E_n の総和を最小化することなので、勾配法、特に最急降下法を用いる。

$$\boldsymbol{w}' = \boldsymbol{w} - \eta \frac{\partial}{\partial \boldsymbol{w}} \sum_{n=1}^{N} E_n \tag{3}$$

各次元の独立を仮定して d 次元目に着目すると、式 (3) は

$$w'_d = w_d - \eta \frac{\partial}{\partial w_d} \sum_{n=1}^{N} E_n = w_d - \eta \sum_{n;E_n \neq 0}^{N} t_n x_{nd} \tag{4}$$

これを全次元で繰り返し実行することにより徐々に重み $\boldsymbol{w}$ が最適化される。では、実装してみよう。

```
  val eta = 0.1
  for(step <- 1 to 10000) samples.foreach{case (in, out) => {
    predict(in) match {
      case true if !out =>
        for(k <- 0 until dim) weights(k + 1) -= eta * in(k)
        weights(0) -= eta * 1
      case false if out =>
        for(k <- 0 until dim) weights(k + 1) += eta * in(k)
        weights(0) += eta * 1
      case _ =>
    }
  }}
```

以上を先ほどの Perceptron クラスのコンストラクタに記述すればよい。論理和を学習させてみよう。

```
val samples = Seq(
  (Seq(0.0, 0.0), false),
  (Seq(0.0, 1.0), true),
  (Seq(1.0, 0.0), true),
  (Seq(1.0, 1.0), true))
val p = new Perceptron(samples, 2);
println(p.predict(Seq(0.0, 0.0))) // false
println(p.predict(Seq(0.0, 1.0))) // true
println(p.predict(Seq(1.0, 0.0))) // true
println(p.predict(Seq(1.0, 1.0))) // true
```

本当に Fig. 5 に示したような直線を学習しているかを確かめるには、weights を表示してみればよい。

```
println(p.weights.mkString(",")) // -0.1,0.1,0.1
```

論理和と同じ要領で論理積を学習させることもできる。しかし、排他的論理和の学習は必ず失敗する。

```
val samples = Seq(
  (Seq(0.0, 0.0), false),
  (Seq(0.0, 1.0), true),
  (Seq(1.0, 0.0), true),
  (Seq(1.0, 1.0), false))
val p = new Perceptron(samples, 2);
println(p.predict(Seq(0.0, 1.0))) // true
println(p.predict(Seq(0.0, 1.0))) // true
println(p.predict(Seq(0.0, 1.0))) // false
println(p.predict(Seq(0.0, 1.0))) // false
```

weights を表示すると、0.0,-0.1,0.0 となり、y 軸そのものを学習してしまっている。考えてみれば当然の話で、排他的論理和は直線で分けることができない。決定境界が直線ないし超平面になる問題を**線型分離可能**と呼ぶ。論理和や論理積は線型分離可能であるが、排他的論理和は線型分離不可能なため単純パーセプトロンでは正しく扱うことができない。

3.2 多層パーセプトロン

単純パーセプトロンでは線型分離不可能な問題を扱うことができないが、単純パーセプトロンを何層も積み重ねることで線型分離不可能な問題にも対応できることが知られている。Fig. 6 に模式図を示す。

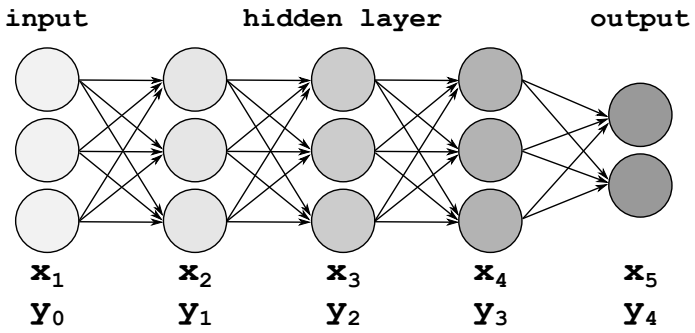

Fig. 6 multi layer perceptron.

単純パーセプトロンをいくつか並列に束ねることで出力 $\boldsymbol{y}$ をベクトルとし、そうしたパーセプトロンを何層にも積み重ねることで**多層パーセプトロン**が構成される。左端で入力ベクトル $\boldsymbol{x}_1$ を受け取る層を**入力層**、右端で出力ベクトル $\boldsymbol{y}_N$ を出力する層を**出力層**、入力層と出力層に挟まれた層を**中間層**と呼ぶ。活性化関数を f とすると、N 層パーセプトロンの最終段の出力 $\boldsymbol{y}_N$ は式 (5) で再帰的に計算される。

$$\boldsymbol{y}_N = f(\boldsymbol{w}_N \cdot \boldsymbol{y}_{N-1}) = f(\boldsymbol{w}_N \cdot f(\boldsymbol{w}_{N-1} \cdot f(\boldsymbol{w}_{N-2} \cdot ...f(\boldsymbol{w}_2 \cdot f(\boldsymbol{w}_1 \cdot \boldsymbol{x}))...)) \tag{5}$$

ただし、便宜的に第 n 段の入力ベクトルを $\boldsymbol{x}_n$、出力ベクトルを $\boldsymbol{y}_n$ と表記するものとする。従って、

$$\boldsymbol{x}_{n+1} = \boldsymbol{y}_n \tag{6}$$

が成立する。活性化関数 f には一般的に式 (7) で定義される標準シグモイド関数を使用する。

$$f_s(x) = \frac{1}{1+e^{-x}} \tag{7}$$

シグモイド関数は階段関数を連続関数で近似したような 0 から 1 へのゆるやかな段差を描く (Fig. 7)。

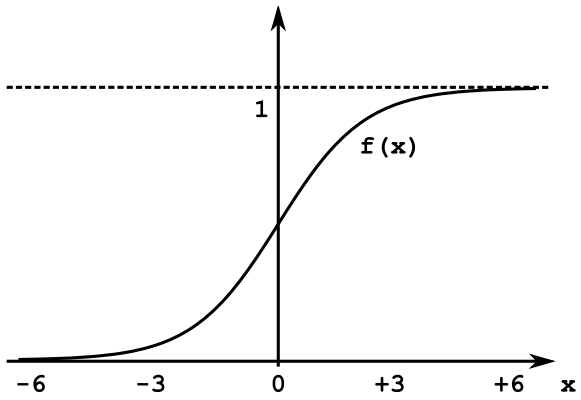

Fig. 7　sigmoid function.

シグモイド関数を利用することで、後述する学習ステップに勾配法を適用する際に、シグモイド関数の導関数がシグモイド関数そのもので表現できるというシグモイド関数の強みが活かせる。

$$\frac{df_s}{dx}(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f_s(x)\{1 - f_s(x)\} \tag{8}$$

今回は中間層が 1 層だけからなる 3 層パーセプトロンを実装しよう。手始めに Data クラスを定義する。

```
case class Data(input: Seq[Double], answer: Seq[Double])
```

例えば、排他的論理和の訓練データは以下のように定義する。

```
val samples = Seq(
  Data(Seq(0.0, 0.0), Seq(0.0)),
  Data(Seq(0.0, 1.0), Seq(1.0)),
  Data(Seq(1.0, 0.0), Seq(1.0)),
  Data(Seq(1.0, 1.0), Seq(0.0)))
```

この訓練データをコンストラクタの引数に取る MLP クラスを定義する。

```
import scala.collection.mutable.Map

class MLP(samples: Seq[Data], I: Int, H: Int, O: Int) {
  val w1 = Map[(Int, Int), Double]()
  val w2 = Map[(Int, Int), Double]()
}
```

I は入力層、H は中間層、O は出力層の階数である。従って、入力層から中間層への伝達では IH 通りの重み $\boldsymbol{w}_{ih}$ が必要で、中間層から出力層への伝達では HO 通りの重み $\boldsymbol{w}_{ho}$ が必要である。学習により重み付けが変更されることを考慮し、可変 Map で重みを保持する。次に中間層と出力層の出力を実装する。

```
def hidden(args: Seq[Double]): Seq[Double] = {
  val hid = new Array[Double](H)
  for(i <- 0 until I; h <- 0 until H) hid(h) += w1((i, h)) * args(i)
  for(h <- 0 until H) hid(h) = sigmoid(hid(h))
  hid
}
```

```
def output(args: Seq[Double]): Seq[Double] = {
  val out = new Array[Double](O)
  for(h <- 0 until H; o <- 0 until O) out(o) += w2((h, o)) * args(h)
  for(o <- 0 until O) out(o) = sigmoid(out(o))
  out
}
```

hidden メソッドと output メソッドは入力としてベクトル $\boldsymbol{x}_n$ を受け取り、ベクトル $\boldsymbol{y}_n$ を出力する。何れも sigmoid 関数を呼び出しているが、式 (7) に示した標準シグモイド関数の定義通りに実装する。

```
def sigmoid(x: Double) = 1.0 / (1.0 + Math.exp(-x))
```

式 (5) に従って入力ベクトルを受け取り出力ベクトルを計算する predict メソッドを定義しておこう。

```
def predict(input: Seq[Double]) = output(hidden(input))
```

以上で、3 層パーセプトロンはクラス分類器として利用可能になった。問題は、どうやって重み付けを学習させるかである。基本的な考え方は単層パーセプトロンと同じで、最終段の出力誤差が最小になる重み付けを勾配法により探索する。ここで、出力誤差を徐々に最前段に伝搬させていく**誤差逆伝搬法**を導入する。最終段の出力 $\boldsymbol{y}_{output}$ と正解 $\boldsymbol{y}_{answer}$ との二乗誤差 E を式 (9) で定義する。

$$E = ||\boldsymbol{y}_{output} - \boldsymbol{y}_{answer}||^2 \tag{9}$$

第 n 番目の層で、i 次元目から j 次元目への経路 $(i \to j)$ の重み w_n^{ij} を最急降下法により最適化する。

$$w_n'^{ij} = w_n^{ij} - \eta \frac{\partial E}{\partial w_n^{ij}} \tag{10}$$

合成関数の微分の公式を用いて式 (10) を展開すると、

$$\frac{\partial E}{\partial w_n^{ij}} = \frac{\partial E}{\partial n_n^{ij}} \frac{\partial n_n^{ij}}{\partial w_n^{ij}} = \frac{\partial E}{\partial n_n^{ij}} x_n^i \tag{11}$$

ただし、x_n^j は第 n 層の入力 $\boldsymbol{x}_n$ の i 次元目で、n_n^{ij} は経路 $(i \to j)$ の寄与 $w_n^{ij} x_n^i$ である。

$$\frac{\partial E}{\partial n_n^{ij}} = \frac{\partial E}{\partial y_n^j} \frac{\partial y_n^j}{\partial n_n^{ij}} = \frac{\partial E}{\partial y_n^j} \frac{\partial}{\partial n_n^{ij}} f_s(n_n^{ij}) \tag{12}$$

シグモイド関数 f_s の定義式 (7) を思い出しつつ第 n 層の出力 $\boldsymbol{y}_n$ の j 次元目を y_n^j と定義すると、

$$\frac{\partial}{\partial n_n^{ij}} f_s(n_n^{ij}) = \frac{e^{-n_n^{ij}}}{(1 + e^{-n_n^{ij}})^2} = y_n^j (1 - y_n^j) \tag{13}$$

ここから、第 $n+1$ 層から第 n 層への誤差逆伝搬の漸化式を導出することができる。

$$\frac{\partial E}{\partial y_n^j} = \sum_{k=1}^{K} \frac{\partial E}{\partial n_{n+1}^{jk}} \frac{\partial n_{n+1}^{jk}}{\partial y_n^j} = \sum_{k=1}^{K} \frac{\partial E}{\partial y_{n+1}^k} y_{n+1}^k (1 - y_{n+1}^k) w_{n+1}^{jk} \tag{14}$$

漸化式 (14) の境界条件として、最終段での二乗誤差の勾配を用いる。

$$\frac{\partial E}{\partial y_{output}^j} = 2(y_{output}^j - y_{answer}^j) \tag{15}$$

最終段の二乗誤差の勾配が出力側から入力側へと漸化式 (14) に従って伝搬していく様子が朧げながら想像できるだろうか。数式を並べた解説は以上で終わりということにして、さっそく実装してみよう。

```
for(i <- 0 until I; h <- 0 until H) w1((i, h)) = 2 * Math.random - 1
for(h <- 0 until H; o <- 0 until O) w2((h, o)) = 2 * Math.random - 1
```

重みを乱数で初期化する作業を忘れずに入れておこう。全ての重みが 0 だと誤差が逆伝搬できない。

```
val eta = 0.0
for(step <- 1 to 10000) samples.foreach{case Data(input, ans) => {
  val hid = hidden(input)
  val out = output(hid)
  for(h <- 0 until H) {
    val e2 = for(o <- 0 until O) yield ans(o) - out(o)
    val g2 = for(o <- 0 until O) yield out(o) * (1 - out(o)) * e2(o)
    for(o <- 0 until O) w2((h, o)) += eta * g2(o) * hid(h)
    val e1 = for(o <- 0 until O) yield g2(o) * w2((h, o))
    val g1 = for(i <- 0 until I) yield hid(h) * (1 - hid(h)) * e1.sum
    for(i <- 0 until I) w1((i, h)) += eta * g1(i) * input(i)
  }
}}
```

以上を先掲の MLP クラスのコンストラクタ内に追記する。さっそく排他的論理和を学習させてみよう。

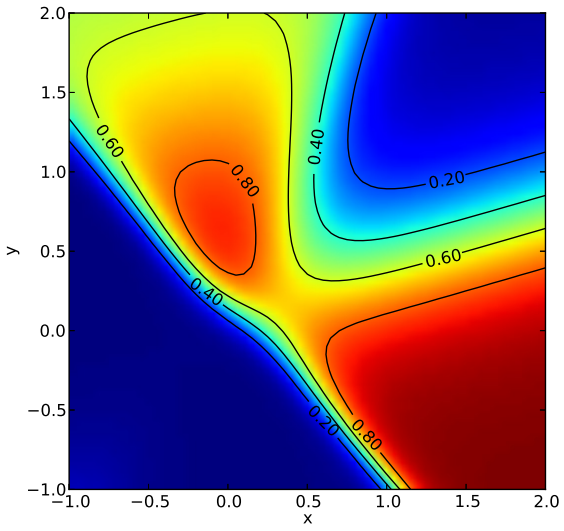

Fig. 8　exclusive or(10,000 steps).

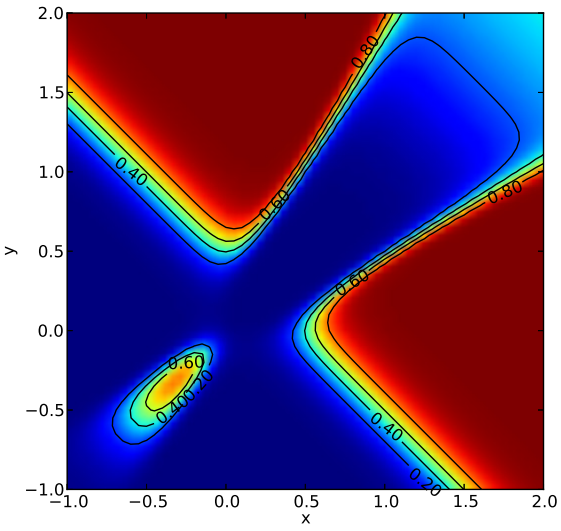

Fig. 9　exclusive or(1,000,000 steps).

Fig. 8 は排他的論理和を値域が $[0,1]$ の実関数と見なして学習させた際の出力値である。活性化関数にシグモイド関数を利用しているので出力値は $[0,1]$ の範囲に収まる。出力として真偽値が欲しい場合は 0.5 を閾値として分離すれば良い。今回は誤差逆伝搬を 10,000 回繰り返す実装としたが、繰り返し数を増やすと徐々に分布が変化する。Fig. 9 は 1,000,000 回繰り返した際の分布である。左下に新たな島が出現するのがわかる。最終的にこの状態で収束するため、誤差が十分に小さくなれば打ち切ると良い。

4 サポートベクターマシン

4.1 ハードマージンの場合

サポートベクターマシンは、ふたつの集団を分離する**識別直線**を学習する機械である。Fig. 10 を見よ。

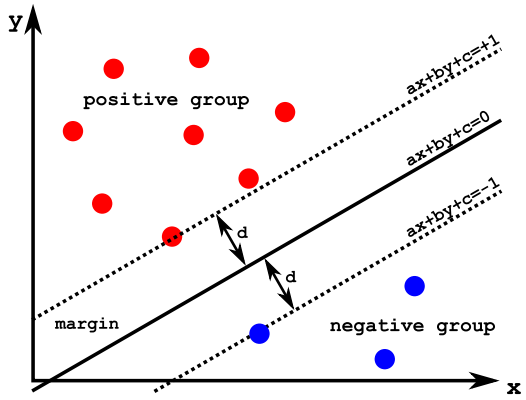

Fig. 10 support vector machine.

サポートベクターマシンが学習するのは単なる決定境界ではない。正集団と負集団からの最短距離 d が最大になる直線を学習するのである。仮に現在知られている訓練データよりも境界に近い部分に未知のデータが出現した場合でも、他方の集団に誤って分類される危険性は低くなる、という期待ができる。既にお気付きの方もいるかもしれないが、これは制約付き最大値問題である。点 (x_i, y_i) の制約条件は、

$$|ax_i + by_i + c| \geq 1 \tag{16}$$

ただし、a, b, c は識別直線の係数で、任意の実数値である。集団と識別直線との最短距離は

$$\min d(x_i, y_i) = \min \frac{|ax_i + by_i + c|}{\sqrt{a^2 + b^2}} = \frac{1}{\sqrt{a^2 + b^2}} \tag{17}$$

あとはラグランジュの未定乗数法を利用して $a^2 + b^2$ を最小化するだけである。正解を $t_i = \pm 1$ として

$$L(a, b, c, \lambda) = \frac{1}{2}(a^2 + b^2) - \sum_{i=1}^{N} \lambda_i \{t_i(ax_i + by_i + c) - 1\} \tag{18}$$

a, b, c で偏微分することにより、L が最小になり最短距離 d が最大になる条件が求まる。

$$a = \sum_{i=1}^{N} \lambda_i t_i x_i,\ b = \sum_{i=1}^{N} \lambda_i t_i y_i,\ 0 = \sum_{i=1}^{N} \lambda_i t_i \tag{19}$$

$$\tilde{L}(\lambda) = \min_{a,b,c} L(a, b, c, \lambda) = \sum_{i=1}^{N} \lambda_i \left\{ 1 - \frac{1}{2} \sum_{j=1}^{N} \lambda_j t_i t_j (x_i x_j + y_i y_j) \right\} \tag{20}$$

あとは双対問題 $\tilde{L}(\lambda)$ を最大化するだけである。最急上昇法を利用する方法が数多く紹介されているが、今回は**逐次最小問題最適化法**を採用する。これまで未定乗数法を利用して議論してきたが、厳密には、式 (16) の制約式は不等式条件であり、KKT(Karush–Kuhn–Tacker) 条件を適用しなければならない。KKT 条件を適用することにより、式 (20) の最適化問題の解が存在する必要条件は、

$$\lambda_i\{t_i(ax_i + by_i + c) - 1\} = 0, \ \ (\lambda_i \geq 0) \tag{21}$$

仮に $\lambda_i > 0$ である場合、$ax_i + by_i + c = t_i$ が成立しなければならない。このことは即ち、(x_i, y_i) が Fig. 10 の点線上に存在することを意味する。逆に $\lambda_i = 0$ である場合、式 (19) を見ればわかるように係数 a, b の決定に何ら寄与しない。従って、サポートベクターマシンの学習には Fig. 10 の識別直線に最も近い位置にある訓練データのみが考慮される。このような点を**サポートベクトル**と呼び、これこそサポートベクターマシンの優れた汎化能力の理由である。さて、逐次最小問題最適化法では式 (21) の条件を破る訓練データ (x_{i1}, y_{i1}) が存在する限り任意の訓練データ (x_{i2}, y_{i2}) を選び、式 (19) を根拠に

$$\lambda_{i1}^{new} t_{i1} + \lambda_{i2}^{new} t_{i2} = \lambda_{i1}^{old} t_{i1} + \lambda_{i2}^{old} t_{i2} \tag{22}$$

を満たす局所的な最適化を施す操作を繰り返す。ここで、1 回の最適化による λ_{i1} と λ_{i2} の増分を

$$\delta_{i1} = \lambda'_{i1} - \lambda_{i1} \tag{23}$$
$$\delta_{i2} = \lambda'_{i2} - \lambda_{i2} \tag{24}$$

と定義して、式 (20) の双対問題 $\tilde{L}(\lambda_{i1}, \lambda_{i2})$ のうち、δ_{i1}, δ_{i2} が関わる部分を抜き出すと、

$$\begin{aligned} d\tilde{L}(\delta_{i1}, \delta_{i2}) = \ & +\delta_{i1} + \delta_{i2} - \delta_{i1}\delta_{i2} t_{i1} t_{i2} (x_{i1} x_{i2} + y_{i1} y_{i2}) \\ & -\delta_{i1} t_{i1} \sum_{n=1}^{N} \lambda_n t_n (x_{i1} x_n + y_{i1} y_n) - \frac{1}{2}\delta_{i1}^2 t_{i1}^2 (x_{i1}^2 + y_{i1}^2) \\ & -\delta_{i2} t_{i2} \sum_{n=1}^{N} \lambda_n t_n (x_{i2} x_n + y_{i2} y_n) - \frac{1}{2}\delta_{i2}^2 t_{i2}^2 (x_{i2}^2 + y_{i2}^2) \end{aligned} \tag{25}$$

ここはぜひ確かめてほしい。$t_{i1} = t_{i2}$ の場合、式 (22) の条件より $\delta_{i1} = -\delta_{i2}$ なので、

$$\begin{aligned} d\tilde{L}(\delta_{i1}) = \ & +\delta_{i1}^2 (x_{i1} x_{i2} + y_{i1} y_{i2}) \\ & -\delta_{i1} t_{i1} \sum_{n=1}^{N} \lambda_n t_n (x_{i1} x_n + y_{i1} y_n) - \frac{1}{2}\delta_{i1}^2 (x_{i1}^2 + y_{i1}^2) \\ & +\delta_{i1} t_{i1} \sum_{n=1}^{N} \lambda_n t_n (x_{i2} x_n + y_{i2} y_n) - \frac{1}{2}\delta_{i1}^2 (x_{i2}^2 + y_{i2}^2) \end{aligned} \tag{26}$$

$\tilde{L}(\lambda)$ を最大化することは式 (25) の極大点を探すことと同じである。式 (25) を δ_{i1} で微分すると、

$$\delta_{i1} = -t_{i1} \frac{\displaystyle\sum_{n=1}^{N} \lambda_n t_n \{(x_{i1} - x_{i2}) x_n + (y_{i1} - y_{i2}) y_n\} - t_{i1} + t_{i2}}{(x_{i1} - x_{i2})^2 + (y_{i1} - y_{i2})^2} \tag{27}$$

が成立するとき式 (25) が極大値になることがわかる。$t_{i1} = -t_{i2}$ の場合も式 (27) が得られる。あえて $t_{i1} - t_{i2}$ という余分な項を追加することで、$t_{i1} = t_{i2}$ の場合でも $t_{i1} \neq t_{i2}$ の場合でも共通の式 (27) が利用できるように工夫されている。訓練データ (x_i, y_i) が式 (21) の条件を満たすかどうかを調べるには係数 a と b と c が必要である。係数 a と b は式 (19) により求まるが、バイアス項 c だけ求められない。だが、Fig. 10 に示したように、$t_i(ax_i + by_i)$ が最小になる (x_i, y_i) はサポートベクトルである可能性が高い。バイアス項 c の値を近似的に求めるには、そのようなサポートベクトルの候補を探し出して、

$$c = -\frac{1}{2}\left\{ \min_{i|t_i=+1} (ax_i + by_i) + \max_{j|t_j=-1} (ax_j + by_j) \right\} \tag{28}$$

とすれば良い。さて、式 (27) を計算すれば λ_{i1} と λ_{i2} の値が更新できるのであるが、ここで油断してはならない。更新後のラグランジュ乗数もまた、式 (21) の条件を満たさなければならない。具体的には更新後の乗数が負の値になってはいけない。従って、$t_{i1} = t_{i2}$ の場合は、

$$\lambda_{i1}^{new} = \lambda_{i1}^{old} + \delta_{i1} \geq 0 \tag{29}$$
$$\lambda_{i2}^{new} = \lambda_{i2}^{old} - \delta_{i1} \geq 0 \tag{30}$$

同様に、$t_{i1} \neq t_{i2}$ の場合は、

$$\lambda_{i1}^{new} = \lambda_{i1}^{old} + \delta_{i1} \geq 0 \tag{31}$$
$$\lambda_{i2}^{new} = \lambda_{i2}^{old} + \delta_{i1} \geq 0 \tag{32}$$

この条件を満たさない場合は δ_{i1} をクリッピングする。クリッピングという用語は、グラフィックスで図形を表示する際に表示領域の外側を切り取ることを表すクリッピングから来ている。例えば、λ_{i1} が負になる場合は、δ_{i1} の値を $\lambda_{i1} = 0$ となるように書き換えれば良い。以上で逐次最小問題最適化法によるサポートベクターマシンの訓練に必要な数式は全て出揃った。さっそくサポートベクターマシンの実装に入ろう。手始めに、1 件の訓練データを表す Data クラスを定義する。

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

t は正解ラベル t_i で、l はラグランジュ乗数 λ_i である。続いて SVM クラスを定義する。

```
class SVM(samples: Seq[Data]) {
  var (a, b, c) = (0.0, 0.0, 0.0)
  def predict(x: Double, y: Double) = a * x + b * y + c > 0
}
```

式 (28) に従って係数 c の近似値を求める newc メソッドを SVM クラスに定義する。

```
  def newc = {
    val pos = samples.filter(_.t == +1).map(d => a * d.x + b * d.y).min
    val neg = samples.filter(_.t == -1).map(d => a * d.x + b * d.y).max
    (pos + neg) / 2
  }
```

式 (19) に従って係数 a と b を求める newa メソッドと newb メソッドも定義する。

```
  def newa = (for (d <- samples) yield d.l * d.t * d.x).sum
  def newb = (for (d <- samples) yield d.l * d.t * d.y).sum
```

これにより、訓練データが式 (16) と式 (21) を満たすことを確認する kkt メソッドが定義できる。

```
  def kkt(d: Data) = d.l match {
    case 0 => d.t * (a * d.x + b * d.y + c) >= 1
    case _ => d.t * (a * d.x + b * d.y + c) == 1
  }
```

ここで、式 (27) に基づいてラグランジュ乗数 λ_{i1} と λ_{i2} を更新する update メソッドを定義する。

```
def update(d1: Data, d2: Data) {
  val (dx, dy) = (d1.x - d2.x, d1.y - d2.y)
  val den = dx * dx + dy * dy
  val num = (for(d <- samples)
    yield d.l * d.t * (dx * d.x + dy * d.y)
  ).sum - d1.t + d2.t
  val di1 = clip(d1, d2, -d1.t * num / den)
  d1.l += di1
  d2.l -= di1 * d1.t * d2.t
}
```

クリッピングを行うための clip メソッドも定義しておく。

```
def clip(d1: Data, d2: Data, di1: Double): Double = {
  if (d1.t == d2.t) {
    if (di1 < -d1.l) return -d1.l
    if (di1 > +d2.l) return +d2.l
    return di1
  } else return Seq(-d1.l, -d2.l, di1).max
}
```

ここまで来れば、式 (21) を満たさない訓練データを見つけて update メソッドに渡すだけである。

```
while (samples.exists(!kkt(_))) {
  val i1 = samples.indexWhere(!kkt(_))
  var i2 = 0
  do i2 = (math.random * samples.size).toInt while(i1 == i2)
  update(samples(i1), samples(i2))
  a = newa
  b = newb
  c = newc
}
```

ついにサポートベクターマシンの完成である。検証を兼ねて Fig. 11 に示すデータを訓練させてみた。

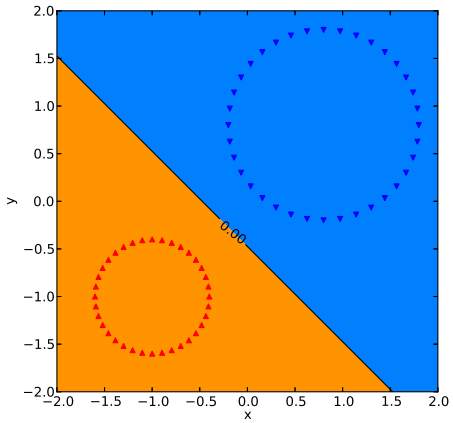

Fig. 11 learning of a hard margin SVM.

ふたつの円から等距離で、中心を結んだ線分と垂直に交差する識別直線が描けていれば大成功である。

4.2　ソフトマージンの場合

第 4.1 節で解説したサポートベクターマシンは、正負の集団を完璧に分離する識別直線に到達するまで学習を継続する。得られた識別直線は一切の誤分類なくデータを正しく正と負に分離する。逆に言えばサポートベクターマシンは誤分類の発生を適切に想定していない。現実世界のデータは大域的に見れば美しい分布を示すように見えても、局所的にはしばしば非理想的な分布を示すことがある。Fig. 12 に示すように、赤い点の集団である正集団に 1 点だけ負の青いデータ点が混ざっている状況もあり得る。

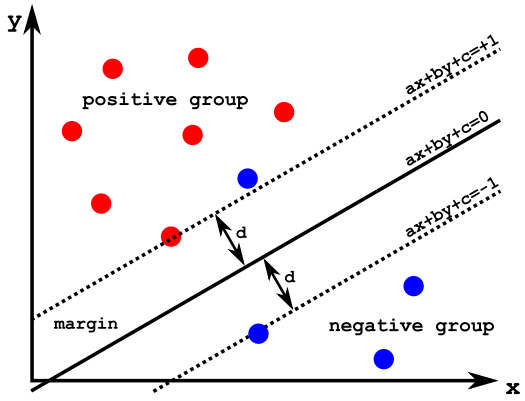

Fig. 12　soft margin SVM.

図の識別直線は青い点を正しく分離できていない。従ってこの青い点は正のデータとして誤分類されてしまう。サポートベクターマシンは強引にでもこの誤分類点を解消しようと試みる。だが、境界付近で正負が入り組んだ分布になっている場合など、必ずしも分離に成功するとは言えない。よしんば完璧に分離する識別直線を学習できたとしても**過学習**に陥っている可能性がある。過学習とは、訓練データの分類精度を限界まで高めようと努力する余り、母集団全体での分類精度がおざなりになっている状況である。言い換えれば、木を見て森を見ずの状況であり、未知のデータを適切に分類できない。過学習はサポートベクターマシンに限らず機械学習全般に共通する問題であるが、対策としては目先の誤分類に囚われず寛容の精神を以て許すことが求められる。サポートベクターマシンの場合は**ソフトマージン**の導入が対策として有効である。ソフトマージンは多少の誤分類ならば許容する識別境界である。ただし誤分類された訓練データ $\boldsymbol{x}_i$ に対しては個別にペナルティ ξ_i を課すものとする。式 (16) の制約条件は

$$t_n(ax_n + by_n + c) \geq 1 - \xi_i \tag{33}$$

と書き換えられる。ペナルティ ξ_i は各訓練データに割り当てられるスラック変数で、次式で定義する。

$$\xi_i = \begin{cases} 0 & if \;\; t_i(ax_i + by_i + c) > 1 \\ |t_i - (ax_i + by_i + c)| \geq 1 & if \;\; t_i(ax_i + by_i + c) \leq 1 \end{cases} \tag{34}$$

最小化すべき目的関数は、定数 $C > 0$ を導入することにより、

$$C\sum_{i=1}^{N} \xi_i + \frac{1}{2}(a^2 + b^2) \tag{35}$$

任意の誤分類データ (x_i, y_i) について $\xi_i \geq 1$ が成立することから、$\sum \xi_i$ は誤分類された訓練データの個数よりも大きな値を取る。即ち、$\sum \xi_i$ は誤分類された個数の上限を与える。このことから定数 C は訓練データでの誤分類をどの程度許容するかを間接的に決定する定数として作用することが期待できる

ように見える。少なくとも $C \to \infty$ の極限では誤分類は全く許容されない。従って第 4.1 節で扱った**ハードマージン**はソフトマージンの特殊な場合と見なすことができる。式 (18) のラグランジュ関数は

$$L(a,b,c,\xi,\lambda,\mu) = \frac{1}{2}(a^2+b^2) + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i\{t_i(ax_i+by_i+c)-1\} - \sum_{i=1}^{N}\mu_i\xi_i \tag{36}$$

と書き換えられる。式 (18) の未定乗数は λ_i だけであったが今回は λ_i と μ_i が未定乗数である。今回も前回と同様に a, b, c で偏微分することにより、L が最小になりマージンが最大化する条件が求まる。

$$a = \sum_{i=1}^{N}\lambda_i t_i x_i,\ b = \sum_{i=1}^{N}\lambda_i t_i y_i,\ 0 = \sum_{i=1}^{N}\lambda_i t_i,\ \lambda_i = C - \mu_i \tag{37}$$

$$\tilde{L}(\lambda) = \min_{a,b,c} L(a,b,c,\lambda) = \sum_{i=1}^{N}\lambda_i\left\{1 - \frac{1}{2}\sum_{j=1}^{N}\lambda_j t_i t_j (x_i x_j + y_i y_j)\right\} \tag{38}$$

式 (38) はハードマージンでの双対形 (20) と同一である。ただし、KKT 条件から導かれる必要条件は

$$\lambda_i\{t_i(ax_i+by_i+c)+\xi_i-1\} = 0,\ \ (\lambda_i \geq 0,\ \ \mu_i \geq 0,\ \ \mu_i\xi_i = 0) \tag{39}$$

と変化することに注意したい。また、式 (37) より次式の制約条件が導出される。

$$0 \leq \lambda_i \leq C \tag{40}$$

以上の議論に基づき逐次最小問題最適化法の適用を試みる。まず、$\lambda_i = 0$ となる訓練データ (x_i, y_i) は式 (38) より $\mu_i = C > 0$ が成立する筈である。式 (39) を満たすには $\xi_i = 0$ が必要である。この条件は ξ_i の定義上 $|ax_i + by_i + c| > 1$ と同等である。次に、$\lambda_i = C$ となる訓練データ (x_i, y_i) は式 (38) より $\mu_i = 0$ が成立する筈である。この際 $\xi_i > 0$ となる可能性があり、式 (39) より $t_i(ax_i + by_i + c) \leq 1$ が成立する。最後に、$0 < \lambda_i < C$ となる訓練データ (x_i, y_i) は $\mu_i > 0$ ゆえ $\xi_i = 0$ が成立する筈である。この訓練データは正しく分類され、しかも $t_i(ax_i + by_i + c) = 1$ の境界線上に存在している筈である。以上の考察より、第 4.1 節で実装した `kkt` メソッドを以下のように書き換える。

```
def kkt(d: Data) = d.l match {
  case 0 => d.t * (a * d.x + b * d.y + c) >= 1
  case C => d.t * (a * d.x + b * d.y + c) <= 1
  case _ => d.t * (a * d.x + b * d.y + c) == 1
}
```

式 (38) は制約条件を除きハードマージンにおける式 (20) と同等であるため、式 (27) に示した最適化の議論をそのまま流用してよい。ただし、式 (40) の制約を守るためにクリッピングが必要になる場合がある。$t_{i1} = t_{i2}$ の場合は、

$$0 \leq \lambda_{i1}^{new} = \lambda_{i1}^{old} + \delta_{i1} \leq C \tag{41}$$
$$0 \leq \lambda_{i2}^{new} = \lambda_{i2}^{old} - \delta_{i1} \leq C \tag{42}$$

同様に $t_{i1} \neq t_{i2}$ の場合は、

$$0 \leq \lambda_{i1}^{new} = \lambda_{i1}^{old} + \delta_{i1} \leq C \tag{43}$$
$$0 \leq \lambda_{i2}^{new} = \lambda_{i2}^{old} + \delta_{i1} \leq C \tag{44}$$

従って、第 4.1 節で定義した `clip` メソッドを以下のように書き換える。

```
  def clip(d1: Data, d2: Data, di1: Double): Double = {
    if (d1.t == d2.t) {
      val L = Math.max(-d1.l, d2.l - C)
      val H = Math.min(+d2.l, C - d1.l)
      if (di1 < L) return L
      if (di1 > H) return H
      return di1
    } else {
      val L = Math.max(-d1.l, -d2.l)
      val H = Math.min(-d1.l, -d2.l) + C
      if (di1 < L) return L
      if (di1 > H) return H
      return di1
    }
  }
```

SVM クラスの冒頭に定数 C を宣言することも忘れずに行っておく。

```
class SVM(samples: Seq[Data]) {
  val C = 0.01
}
```

以上でソフトマージンの実装は完了である。検証のため Fig. 13 に示すデータを訓練させてみた。

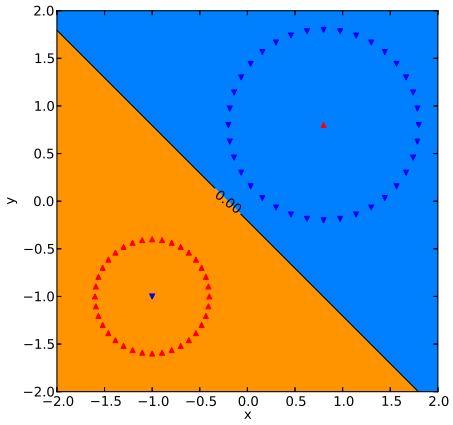

Fig. 13　learning of a soft margin SVM.

定数 C の値を変化させると学習の様子がどのように変化するかを観察してみると面白い。大きな値に設定すると学習が終わらなくなり無限ループに陥る。小さな値に設定すれば誤分類データは見捨てられ学習は早期に終結する。Fig. 13 の例は少々単純すぎるので境界部分が複雑に入り組んだ訓練データを用意するのも良いだろう。興味深いことに Fig. 13 は Fig. 11 と比べて識別直線が負の側に誘引されている。実はサポートベクターマシンには学習の結果が**外れ値**に左右されやすいという難点がある。図の例ではふたつの円の中心にある誤分類点が外れ値であるが、特に境界から遠方にある右上の誤分類点の方向へ識別直線が引き寄せられている。結果として訓練データに対する分類能力はこの正負の外れ値を除き正確に保たれるが、識別直線の近傍に出現する未知データに対する分類精度は低下する恐れがある。

5 カーネルトリック

第 4 章で解説したサポートベクターマシンは多層パーセプトロンとは異なり線型分離不可能な問題には適用できない。だが、現にサポートベクターマシンは線型分離不可能な問題にも適用されており、最も普及した機械学習アルゴリズムのひとつになっている。実は、線型分離不可能な問題を線型分離可能な問題へと変えてしまう便利な方法があるのだ。本章ではその理論的背景とコーディング例を紹介する。

5.1 高次元空間への写像

Fig. 14 は線型分離不可能な問題を線型分離可能にする原理を模式的に表したものである。

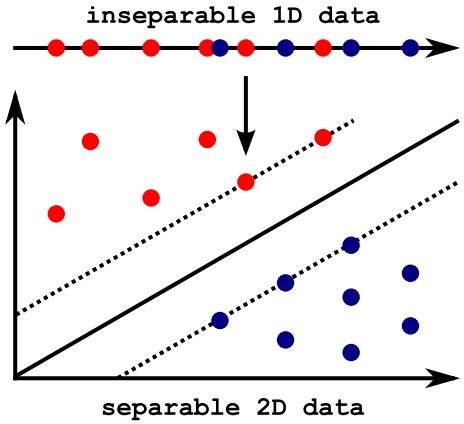

Fig. 14 conversion of linearly inseparable problem to separable.

線型分離不可能な問題は高次元空間に写像すれば線型分離可能になる可能性がある。次元が増大するに伴い訓練データの分布は粗になり、ついには線型分離可能問題に落ち着く。一般に n 個の訓練データがある場合 $n-1$ 次元の空間にまで写像すれば必ず線型分離できる。そこまでではなくとも、ある程度の次元にまで上げてしまえば線型分離できる場合も多々ある。d 次元ベクトルを $d'(>d)$ 次元ベクトルに変換する処理を噛ませれば、何れにせよ、サポートベクターマシンでも線型分離不可能な問題を扱える可能性が高まるはずである。以上の確信に基づき、第 4.2 節までに実装したサポートベクターマシンを改造する。手始めに高次元空間 Ω への写像 Φ を定義する。写像 Φ の内容については後述する。

$$\Phi : \boldsymbol{x} \mapsto \Phi_{\boldsymbol{x}} \quad (\Phi_{\boldsymbol{x}} \in \Omega) \tag{45}$$

第 4 章で導出した式は訓練データ (x_i, y_i) を $\Phi_{\boldsymbol{x}}$ に置き換えることで本章の議論にそのまま流用できる。なお、今後は訓練データ (x_i, y_i) をベクトル $\boldsymbol{x}$ で、係数 a 及び b をベクトル $\boldsymbol{w}$ で表現することにする。逐次最小問題最適化法で直接計算する必要がある式のみ抽出して列挙すると、式 (39)(27)(28) は

$$\lambda_i\{t_i(\boldsymbol{w}\cdot\Phi_{\boldsymbol{x}}+c)+\xi_i-1\}=0, \quad (\lambda_i \geq 0, \ \ \mu_i \geq 0, \ \ \mu_i\xi_i=0) \tag{46}$$

$$\delta_{i1} = -t_{i1}\frac{\sum_n \lambda_n t_n \langle \Phi_{\boldsymbol{x}_{i1}} - \Phi_{\boldsymbol{x}_{i2}}, \Phi_{\boldsymbol{x}_n}\rangle - t_{i1} + t_{i2}}{\langle \Phi_{\boldsymbol{x}_{i1}}, \Phi_{\boldsymbol{x}_{i1}}\rangle - 2\langle \Phi_{\boldsymbol{x}_{i1}}, \Phi_{\boldsymbol{x}_{i2}}\rangle + \langle \Phi_{\boldsymbol{x}_{i2}}, \Phi_{\boldsymbol{x}_{i2}}\rangle} \tag{47}$$

$$c = -\frac{1}{2}\left\{\min_{i|t_i=+1} \boldsymbol{w}\cdot\Phi_{\boldsymbol{x}_i} + \max_{j|t_j=-1} \boldsymbol{w}\cdot\Phi_{\boldsymbol{x}_j}\right\} \tag{48}$$

$\langle \cdot, \cdot \rangle$ は内積である。式 (37) より係数 $\boldsymbol{w}$ が全訓練データに対する総和で計算できることを思い出そう。

$$\boldsymbol{w} = \sum_{i=1}^{N} \lambda_i t_i \Phi_{\boldsymbol{x}_i} \tag{49}$$

従って、式 (46) と式 (48) は以下のように内積を用いて再定義できる。

$$\lambda_i \left\{ t_i \left(\sum_{j=1}^{N} \lambda_j t_j \langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle + c \right) + \xi_i - 1 \right\} = 0 \tag{50}$$

$$c = -\frac{1}{2} \left\{ \min_{i|t_i=+1} \sum_{j=1}^{N} \lambda_j t_j \langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle + \max_{j|t_j=-1} \sum_{i=1}^{N} \lambda_i t_i \langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle \right\} \tag{51}$$

後述するように、内積を用いることで**カーネルトリック**と呼ばれる技法の恩恵を享受できる。そもそも高次元空間への写像 Φ はサポートベクターマシンの学習に大きな計算量を要求する。訓練データ $\boldsymbol{x}$ を高次元ベクトル $\Phi_{\boldsymbol{x}}$ に変換したことが原因である。カーネルトリックは高次元空間 Ω で行うべき計算を元の空間での計算で代用することを可能にする。ただし、代用できるのは内積 $\langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle$ に限られる。

5.2 ヒルベルト空間

カーネル法の議論をする際には数学的な議論が必要になり馴染みのない大量の学術用語に困惑するかもしれない。学生は講師がどんな目的意識で理論を持ち出すのかが理解できないので、講義を真面に聞くことを諦めるだろう。本節は厳密ではないながらも議論の流れを追える程度に整理しつつカーネル法の理解を目指す。まず、距離が定義された空間である**距離空間**を導入する。これはユークリッド空間から距離の概念のみを抽出した概念である。距離を定義する**距離関数** d は以下の性質を満たす関数である。

$$\begin{cases} d(x,y) & \geq & 0 \\ x = y & \Leftrightarrow & d(x,y) = 0 \\ d(x,y) & = & d(y,x) \\ d(x,y) & \leq & d(x,z) + d(z,y) \end{cases} \tag{52}$$

式 (52) の性質はそれぞれ**正定値性**、**非退化性**、**対称性**、**劣加法性**と呼ばれる。さて、距離空間ではまだ抽象的すぎるので、制約を加えて**完備距離空間**を定義する。距離空間 M が完備であるとは、M 内部の任意の**コーシー列**が M に属する点に収斂する様子である。コーシー列は次式の制約を満たす列である。

$$\lim_{n,m \to \infty} ||\boldsymbol{x}_n - \boldsymbol{x}_m|| = 0 \tag{53}$$

完備距離空間とは別に**内積空間**も定義する。内積空間もしくは計量ベクトル空間とは、文字通り内積を備えた空間である。ただし、より抽象化された内積である。内積の定義前に、まず**対称半双線型形式**を定義する。対称半双線型形式ないしエルミート形式とは、以下の 2 性質を満たす写像 $\langle \cdot, \cdot \rangle$ である。

$$\begin{cases} \langle \boldsymbol{x}, a\boldsymbol{y} + \boldsymbol{z} \rangle & = & a\langle \boldsymbol{x}, \boldsymbol{y} \rangle + \langle \boldsymbol{x}, \boldsymbol{z} \rangle \\ \langle \boldsymbol{x}, \boldsymbol{y} \rangle & = & \overline{\langle \boldsymbol{y}, \boldsymbol{x} \rangle} \end{cases} \tag{54}$$

前者は線型性であり、後者は**エルミート対称性**である。更に以下の 2 性質を加えたものが内積である。

$$\begin{cases} \langle \boldsymbol{x}, \boldsymbol{x} \rangle & \geq & 0 \\ \langle \boldsymbol{x}, \boldsymbol{x} \rangle = 0 & \Rightarrow & \boldsymbol{x} = 0 \end{cases} \tag{55}$$

前者は正定値性であり、後者は非退化性である。高校数学の範囲で習う**標準内積**はこの定義を満たす。

$$\boldsymbol{x} \cdot \boldsymbol{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i \tag{56}$$

また、関数 f と g を元とする空間 Ω における内積 $\langle f, g \rangle$ は測度を μ とすると次式で定義される。

$$\langle f, g \rangle = \int_{\Omega} f(\boldsymbol{x}) \overline{g(\boldsymbol{x})} d\mu(\boldsymbol{x}) \tag{57}$$

さて、完備距離空間と内積空間を定義したことで、これら双方を満たすクラスである**ヒルベルト空間**の定義が可能になる。ヒルベルト空間 H とは、内積空間でありかつ完備な距離空間であるような空間である。実のところ、ヒルベルト空間は何か特殊な空間というわけではなく、我々が生活するこの空間を抽象化した概念であるに過ぎない。重要なのは、内積と距離が定義された空間ということである。

5.3 再生核ヒルベルト空間

次式のように関数 f を入力に取り関数 T_f を出力するような変換 T を**積分変換**と呼ぶ。

$$T_f(\boldsymbol{s}) = \int_{\Omega_t} k(\boldsymbol{s}, \boldsymbol{t}) f(\boldsymbol{t}) d\boldsymbol{t}, \quad (\boldsymbol{s} \in \Omega_s) \tag{58}$$

変換前の関数 f は $\boldsymbol{t}$ を引数とする関数だが変換後の関数 T_f は $\boldsymbol{s}$ を引数とする関数である。積分変換は領域 Ω_t での計算が難しい問題を別の領域 Ω_s に写像して解き、結果を Ω_s から Ω_t に逆変換することで容易に求解するテクニックで多用される。フーリエ変換 $\mathcal{F}$ やラプラス変換 $\mathcal{L}$ 等は積分変換の代表的な適用例であろう。領域 Ω_t から領域 Ω_s への変換を行う橋渡し的な存在である関数 $k(\boldsymbol{s}, \boldsymbol{t})$ を**カーネル**と呼ぶ。フーリエ変換の場合は $e^{-j\omega t}$ が、ラプラス変換の場合は e^{-st} が該当する。中でも特に**再生性**を有するカーネルを**再生核**と呼ぶ。再生性とは次式のように変換後も同じ関数 f が出現する性質である。

$$f(\boldsymbol{s}) = \int_{\Omega_t} k(\boldsymbol{s}, \boldsymbol{t}) f(\boldsymbol{t}) d\boldsymbol{t} \tag{59}$$

再生核を備えたヒルベルト空間を特に**再生核ヒルベルト空間**と呼ぶ。再生核を使った写像 Φ を考える。

$$\Phi : \boldsymbol{x} \mapsto k(\boldsymbol{x}, \boldsymbol{x}_i) \tag{60}$$

$\boldsymbol{x}_i$ は訓練データの 1 件とする。この写像 Φ を使って訓練データ $\boldsymbol{x}_j$ を高次元ベクトル $\Phi_{\boldsymbol{x}_j}$ に写像する。変換後のベクトル $\Phi_{\boldsymbol{x}_j}$ の線形結合 $f(\boldsymbol{x}_i)$ で構成するベクトル空間は再生核ヒルベルト空間 H_k になる。

$$f(\boldsymbol{x}_i) = \sum_{j=1}^{N} a_j \Phi_{\boldsymbol{x}_j} = \sum_{j=1}^{N} a_j k(\boldsymbol{x}_j, \boldsymbol{x}_i) \tag{61}$$

再生核ヒルベルト空間 H_k 内でベクトル $\Phi_{\boldsymbol{x}_i}$ とベクトル $\Phi_{\boldsymbol{x}_j}$ との内積 $\langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle$ を考えてみよう。

$$\langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle = \langle k(\boldsymbol{x}_i, \boldsymbol{x}_l), k(\boldsymbol{x}_l, \boldsymbol{x}_j) \rangle \tag{62}$$

ここで関数を元に持つ空間での内積の定義を思い出そう。式 (57) より、

$$\langle k(\boldsymbol{x}_i, \boldsymbol{x}_l), k(\boldsymbol{x}_l, \boldsymbol{x}_j) \rangle = \int_{\boldsymbol{x}_l} k(\boldsymbol{x}_i, \boldsymbol{x}_l) k(\boldsymbol{x}_l, \boldsymbol{x}_j) d\boldsymbol{x}_l \tag{63}$$

カーネル k は再生核なので式 (59) の再生性に従い次式が導かれる。

$$\int_{\boldsymbol{x}_l} k(\boldsymbol{x}_i, \boldsymbol{x}_l) k(\boldsymbol{x}_l, \boldsymbol{x}_j) d\boldsymbol{x}_l = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{64}$$

以上の議論により写像 Φ による変換後のベクトル間の内積は写像を経由せずに求まることが判明した。

$$\langle \Phi_{\boldsymbol{x}_i}, \Phi_{\boldsymbol{x}_j} \rangle = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{65}$$

この結論は、単に高次元空間での計算が不要になるという以上の恩恵を我々にもたらす。カーネル法の利用者が写像 Φ の内容を意識する必要性は完全に排除されてしまった。Φ が定義しにくい状況下でもカーネルさえ定義すれば高次元空間に写像できるのだ。さて、以上でカーネルトリックの理論的背景は概ね網羅した。この辺りで具体的に既知のカーネルを見ておこう。まず、**線型カーネル**を紹介する。

$$k_l(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i \cdot \boldsymbol{x}_j = \sum_{d=1}^{D} x_{id} x_{jd} \tag{66}$$

大層な名前が付いているが、要するに標準内積である。既にお気付きだろうが、線型カーネルとは即ち第 4 章で扱った線型サポートベクターマシンそのものである。続いて、**多項式カーネル**を紹介する。

$$k_p(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(a \sum_{d=1}^{D} x_{id} x_{jd} + b \right)^p \tag{67}$$

多項式カーネルの利点は写像後の高次元空間に特徴量の積が出現することである。これにより特徴量の組合わせを新たに特徴量にできる。例えば 2 次元ベクトル $\boldsymbol{x}, \boldsymbol{x}'$ に対し $(\boldsymbol{x} \cdot \boldsymbol{x}')^2$ の基底は次式になる。

$$x_1^2, \ \sqrt{2} x_1 x_2, \ x_2^2 \tag{68}$$

続いて、汎用的に用いられる**ラジアル基底関数**ないし**ガウシアンカーネル**を紹介する。

$$k_{rbf}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{|\boldsymbol{x}_i - \boldsymbol{x}_j|^2}{2\sigma^2} \right) \tag{69}$$

ガウシアンカーネルの特徴は無限次元の特徴空間への写像を表すことにある。級数展開してみよう。

$$\frac{|\boldsymbol{x}_i - \boldsymbol{x}_j|^2}{2\sigma^2} = + \frac{\boldsymbol{x}_i \cdot \boldsymbol{x}_i}{2\sigma^2} - \frac{\boldsymbol{x}_i \cdot \boldsymbol{x}_j}{\sigma^2} + \frac{\boldsymbol{x}_j \cdot \boldsymbol{x}_j}{2\sigma^2} \tag{70}$$

$$\exp \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\sigma^2} = \exp \sum_{d=1}^{D} \frac{x_d y_d}{\sigma^2} = \prod_{d=1}^{D} \exp \frac{x_d y_d}{\sigma^2} \tag{71}$$

$$\exp \frac{x_d y_d}{\sigma^2} = \sum_{n=0}^{\infty} \frac{1}{\sqrt{n!}} \left(\frac{x_d}{\sigma} \right)^n \frac{1}{\sqrt{n!}} \left(\frac{y_d}{\sigma} \right)^n \tag{72}$$

式 (72) は下式の要素を持つ無限次元のベクトルの内積を表現することになる。

$$\frac{1}{\sqrt{n!}} \left(\frac{x_d}{\sigma} \right)^n, \ \ \frac{1}{\sqrt{n!}} \left(\frac{y_d}{\sigma} \right)^n \ \ (n \geq 0) \tag{73}$$

シグモイドカーネルは厳密にはヒルベルト空間を構成しないものの広く使われるカーネルである。

$$k_s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh\left(a \sum_{d=1}^{D} x_{id} x_{jd} + b \right) \tag{74}$$

このカーネルをサポートベクターマシン $\mathrm{sign}(\boldsymbol{w} \cdot \Phi_{\boldsymbol{x}} + c)$ に適用することにより 3 層パーセプトロンを模倣できる。入力層では未知データ $\boldsymbol{x}$ を受け取り、中間層では内積 $\boldsymbol{x} \cdot \boldsymbol{x}_i$ に式 (7) のシグモイド関数を適用し、出力層では重み $\lambda_i t_j$ の線型和に定数項 c を足して符号を確認する。ぜひ確かめてみて欲しい。

5.4 サポートベクターマシンへの適用

これまでの議論を踏まえてサポートベクターマシンにカーネル法を導入して線型分離不可能な問題にも適用できるように改良する。序でながら、これまでの実装が 2 次元空間にしか対応していなかったのを任意次元の特徴空間にも対応できるようにも改良する。従って Data クラスから作り直す必要がある。

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

SVM クラスを任意のカーネルに対応させるためコンストラクタ引数にカーネル k を追加する。第 4 章の実装では係数 a, b を変数として宣言していたが、今回の実装では使用しない。従って、SVM クラスのフィールドには未定乗数 L とソフトマージン係数 C、定数項 c のみを宣言する。

```
class SVM(samples: Seq[Data], k: (Data, Data) => Double) {
  val L = new Array[Double](samples.size)
  val C = 1e100
  var c = 0.0
}
```

この SVM クラスにメソッドを追加していく。まず、$\boldsymbol{w} \cdot \Phi_{\boldsymbol{x}}$ を計算する wx メソッドを定義する。

```
  def wx(x: Data) = (for (s <- samples) yield s.l * s.t * k(x, s)).sum
```

wx メソッドを利用するように kkt メソッドを定義する。

```
  def kkt(d: Data) = d.l match {
    case 0 => d.t * (wx(d) + c) >= 1
    case C => d.t * (wx(d) + c) <= 1
    case _ => d.t * (wx(d) + c) == 1
  }
```

最小問題最適化を実行する update メソッドを第 4.2 節の clip メソッドを流用しつつ定義する。

```
  def update(d1: Data, d2: Data) {
    val den = k(d1, d1) - 2 * k(d1, d2) + k(d2, d2)
    val num = wx(d1 - d2) - d1.t + d2.t
    val di1 = clip(d1, d2, -d1.t * num / den)
    d1.l += di1
    d2.l -= di1 * d1.t * d2.t
  }
```

式 (48) を用いて定数項 c を計算する newc メソッドを定義する。

```
  def newc = {
    val pos = samples.filter(_.t == +1).map(wx(_)).min
    val neg = samples.filter(_.t == -1).map(wx(_)).max
    (pos + neg) / 2
  }
```

最後に、$\boldsymbol{w} \cdot \Phi_{\boldsymbol{x}} + c > 0$ を計算する predict メソッドを定義する。

```
def predict(p: Data) = wx(p) + c > 0
```

ここまで定義すれば、後は SVM クラスのコンストラクタで学習の処理を記述するだけである。

```
while (samples.exists(!kkt(_))) {
  val i1 = samples.indexWhere(!kkt(_))
  var i2 = 0
  do i2 = (math.random * samples.size).toInt while(i1 == i2)
  update(samples(i1), samples(i2))
  c = newc
}
```

試しに式 (69) のガウシアンカーネルを用いて Fig. 15 に示すようなデータを訓練させてみた。

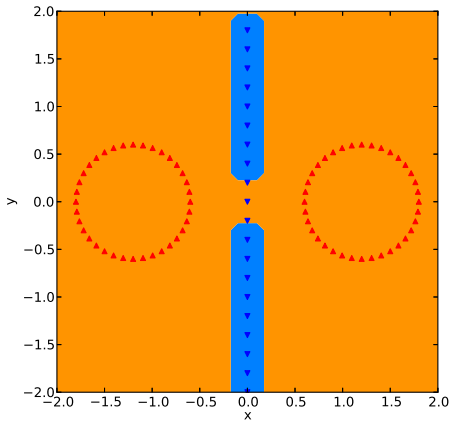

Fig. 15　soft margin SVM with a gaussian kernel.

サポートベクターマシンの適用範囲を飛躍的に広げるカーネル法の偉大さが実感できる。

6　単純ベイズ分類器

本稿はこれまで特徴空間に分布するデータを何らかの関数により分離する方式のアルゴリズムについて議論してきた。今回は視点を変え、条件付き確率を用いて結果から原因を推論する**単純ベイズ分類器**を実装する。単純ベイズ分類器は文書分類アルゴリズムの初歩的な例である。スパムメールフィルタにも応用できる。まず、文字列 D を単語 w の集合とし、文字列 D がクラス C に分類される確率を求める。

$$P(D|C) = P(w_1, \ldots, w_n|C) = \prod_{i=1}^{n} P(w_i|C) \tag{75}$$

本来ならば単語間の**共起関係**も考慮した次式のような複雑な膨大で際限のない確率計算が要求される。

$$\begin{aligned} &P(w_1, \ldots, w_n|C) = P(w_1|C)P(w_2, \ldots, w_n|C, w_1), \\ &P(w_2, \ldots, w_n|C, w_1) = P(w_2|C, w_1)P(w_3, \ldots, w_n|C, w_1, w_2), \\ &P(w_3, \ldots, w_n|C, w_1, w_2) = P(w_3|C, w_1, w_2)P(w_4, \ldots, w_n|C, w_1, w_2, w_3), \ldots \end{aligned} \tag{76}$$

単純ベイズ分類器は単語間の独立性を仮定することで確率計算を大胆に単純化している。文字列 D をクラス C が生成する確率 $P(D|C)$ を定式化したところで、次に、文字列 D の原因として最も相応しいクラス C_j を推測する方法を考える。直観的には、そのようなクラスは $P(C|D)$ を最大にする。

$$j = \arg\max_k P(C_k|D) = \arg\max_k \frac{P(D|C_k)P(C_k)}{P(D)} \tag{77}$$

$P(D)$ が C_k に依存しないことに注意すると、

$$j = \arg\max_k P(D|C_k)P(C_k) \tag{78}$$

式 (75) の仮定より単純ベイズ分類器の分類規則として次式が導かれる。

$$j = \arg\max_k P(C_k)\prod_{i=1}^{n} P(w_i|C_k) \tag{79}$$

観測不可能な潜在事象 A が原因で観測データ B が得られたとき、条件付き確率 $P(B|A)$ は B の原因が事象 A であるという仮説の尤もらしさを表現する。このような確率 $P(B|A)$ を**尤度**と呼ぶ。事象 A の確率分布 $P(A)$ の推定値は観測データ B を得る前と得た後で変化する。B を得る前の推定値 $P(A)$ を**事前確率**と呼び、B を得た後の推定値 $P(A|B)$ を**事後確率**と呼ぶ。クラス C_k を原因として単語 w_i と文字列 D が生成されたと考えれば、$P(w_i|C_k)$ と $P(D|C_k)$ は尤度、$P(C_k|D)$ は事後確率に相当する。従って単純ベイズ分類器は**事後確率最大化法**の応用と見なせる。そろそろ `Data` クラスの実装に入ろう。

```
case class Label(id: String)
case class Data(words: Seq[String], label: Label)
```

可読性のため `Label` クラスを宣言した。続いて分類器の本体である `NaiveBayes` クラスを宣言する。

```
import scala.collection.mutable.{Map, Set}

class NaiveBayes(samples: Seq[Data]) {
  val labels = Set[Label]()
  val vocab = Set[String]()
  val prior = Map[Label, Int]()
  val denom = Map[Label, Int]()
  val numer = Map[(String, Label), Int]()
}
```

`vocab` は訓練データ全体に含まれる互いに異なる単語の集合である。`prior` は事前分布 $P(C)$ を数える可変 `Map` である。`denom` はクラス毎の単語数を単語の重複を許して数える。`numer` は単語とクラスの結合事象について出現回数を数える。条件付き確率 $P(w|C)$ は以下の `pwc` メソッドで計算すれば良い。

```
  def pwc(w: String, c: Label) = numer((w, c)) / denom(c)
```

分類段階では単語 `w` は未知の単語である場合があり、上の定義では $P(w|C) = 0$ となる。その場合でも他の単語では $P(w|C) > 0$ となるだろうから、たとえ未知の単語が出現したとしても既知の単語だけでクラス C を推定できる可能性がある。そこで、出現回数に下駄を履かせて 0 にならないよう工夫する。

$$P(w|C) = \frac{\mathrm{numer}(w \cap C) + 1}{\sum_{w \in vocab} \mathrm{numer}(w \cap C) + 1} = \frac{\mathrm{numer}(w \cap C) + 1}{\mathrm{denom}(C) + |\mathrm{vocab}|} \tag{80}$$

式 (80) は**ラプラススムージング**と呼ばれる。`pwc` メソッドは以下のように書き換える。

```
def pwc(w: String, c: Label) = numer.get((w, c)) match {
  case Some(num) => (num + 1.0) / (denom(c) + vocab.size)
  case None      =>        1.0  / (denom(c) + vocab.size)
}
```

式 (75) に従い $P(D|C)P(C)$ を計算する pdc メソッドを実装する。数値計算の都合により対数とする。

```
def pdc(words: Seq[String], label: Label): Double = {
  val pc = Math.log(prior(label).toDouble / samples.size)
  return pc + words.map(w => Math.log(pwc(w, label))).sum
}
```

$P(D|C)P(C)$ が最大になるクラス C を特定する predict メソッドを実装する。

```
def predict(words: Seq[String]) = labels.maxBy(pdc(words, _))
```

後はコンストラクタに学習の処理を記述するだけである。単語やクラスの出現回数を数えれば良い。

```
samples.foreach{case Data(words, label) => {
  labels += label
  words.foreach(w => numer((w, label))  = 0)
}}
labels.foreach(prior(_) = 0)
labels.foreach(denom(_) = 0)
samples.foreach{case Data(words, label) => {
  vocab ++= words
  prior(label) += 1
  words.foreach(w => denom(label) += 1)
  words.foreach(w => numer((w, label)) += 1)
}}
```

完成した単純ベイズ分類器を使って遊んでみよう。訓練するにも分類するにも何らかの文字列データが必要である。そこで、某百科事典の XML データを利用することにした。文字列を単語に分解するには**形態素解析**ライブラリが便利である。日本語向け形態素解析ライブラリとしては ChaSen や MeCab が著名だが、今回は Lucene GoSen[*1]を利用する。100%Java なライブラリならば Scala で利用するとき何かと使い勝手が良い。形態素解析には辞書データが欠かせないが IPA 辞書を内蔵する jar ファイルを選べば心配無用である。入手した jar ファイルにクラスパスを通して以下のコードを実行してみよう。

```
import net.java.sen.SenFactory
import net.java.sen.dictionary.Token
val tagger = SenFactory.getStringTagger(null)
val tokens = new java.util.ArrayList[Token]()
val source = Source.fromFile("something.txt")
tagger.analyze(source.mkString, tokens)
```

実行すると tokens に単語列が格納される。形態素の情報を取り出すには以下のように記述すれば良い。

[*1] http://github.com/lucene-gosen/lucene-gosen

```
import collection.JavaConversions._
for (morph <- tokens.map(_.getMorpheme)) {
  val ps = morph.getPartOfSpeech
  val surface = morph.getSurface
}
```

記事の XML データから本文を取り出すには scala.xml.XML クラスが便利である。

```
val xml = XML.loadString(source.mkString)
val txt = (xml \\ "text")(0).text
```

形態素解析にかけると、意味のない品詞が多く含まれていることに気付く。文章の内容を表現する上で副詞や助詞、助動詞はほとんど役には立たず、むしろノイズになる。副詞や助詞、助動詞など文法的な機能を果たすものの語彙的な意味を持たない語を**機能語**と呼ぶ。これに対し、名詞や形容詞、動詞など語彙的な意味を持つ語を**内容語**と呼ぶ。情報検索やテキストマイニングでは、機能語を除去することで探索空間を小さくするのは常套手段である。今回は更に限定して固有名詞だけを単語として抽出した。

Table 1　training samples.

山形県	328 語
新潟県	424 語
群馬県	679 語
静岡県	592 語
宮崎県	525 語

Table 1 の記事を訓練データにして Table 2 の火山の所在県を単純ベイズ分類器で推測してみた。

Table 2　classification samples.

赤城山	143 語	群馬県
天城山	89 語	静岡県
榛名山	64 語	群馬県
霧島山	127 語	宮崎県
妙高山	57 語	新潟県
鳥海山	171 語	山形県

霧島山や鳥海山など複数県に跨がる山がどちらの県に分類されるかで遊ぶと面白いかもしれない。

7　最尤推定法

7.1　混合分布モデル

近年は大規模データ活用の社会的需要の高まりを受けて、膨大な観測データから有用な情報を抽出するデータマイニングが注目されている。手法としてはパーセプトロンやサポートベクターマシンのような**線型分類器**を用いる場合もあるが、データが正規分布やポアソン分布などの確率分布に従うと仮定して

平均や分散といったパラメータを学習する統計的手法も有用である。学習した確率分布と観測データを比較することで異常値を検出できる。第 7 章で扱う**混合分布モデル**の場合はデータが観測された原因を推測することもできる。混合分布モデルでは複数の確率分布の線型和により分布が与えられる。例えば Fig. 16 に示す**混合正規分布モデル**では式 (81) のような K 個の正規分布の線型和が確率分布を与える。

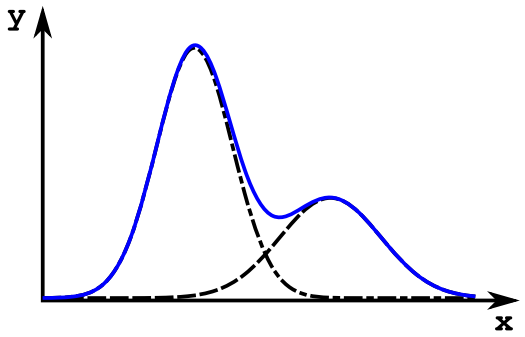

Fig. 16 gaussian mixture model.

$$P(\boldsymbol{x}) = \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, S_k) \quad \left(\sum_{k=1}^{K} w_k = 1 \right) \tag{81}$$

ただし、$\boldsymbol{\mu}$ は平均で、S は分散共分散行列である。混合分布モデルを構成する個別の分布は、モデルの内側の何らかの状態の現れであると見ることができる。つまり、観測者からは見えないもののモデルの内部に K 個の状態が潜在し、重み w_k で状態 k が選択されて $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, S_k)$ に従うデータを生成すると解釈する。例えば、ある高速道路を走行している自動車の速度分布は高速・中速・低速の各潜在状態の確率分布の重ね合わせになる。このように隠れた変数が存在する確率モデルを**潜在変数モデル**と呼ぶ。

7.2 最尤推定の理論

混合分布モデルを使うには各状態のパラメータ θ_k の学習が必須である。混合正規分布モデルの例では $\theta_k = (w_k, \boldsymbol{\mu}_k, S_k)$ を学習により求める必要がある。直観的には尤度 $\mathcal{L}$ を最大にする θ を求めれば良い。

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} P(\boldsymbol{x}_i|\theta) \tag{82}$$

第 6 章で紹介したように、尤度とは潜在事象 A が原因となって観測データ B が得られたという仮説の尤もらしさを表現する量 $P(B|A)$ である。観測データはベクトル $\boldsymbol{x}_i$ の列なので尤度は式 (82) のように同時確率で定義される。混合正規分布モデルの例で具体的な式を示すと、尤度 $\mathcal{L}$ は次式で定義される。

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k, S_k) \tag{83}$$

もっとも、数値計算の都合により、尤度を対数で定義することの方が多い。

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k, S_k) \tag{84}$$

尤度 $\mathcal{L}$ を最大にする θ を探索することにより確率分布のパラメータを学習する手法を**最尤推定**と呼ぶ。目的関数 $\mathcal{L}$ を最大化する θ を計算するのであるから、微分積分学の知識を利用すれば解ける筈である。

しかし残念ながら式 (84) を最大化する $\theta = \{w_k, \boldsymbol{\mu}_k, S_k\}$ を解析的に求めることは容易な仕事ではない。

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_k} = \frac{\partial}{\partial \boldsymbol{\mu}_k} \sum_{i=1}^{N} \log \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, S_k) = \sum_{i=1}^{N} \gamma_{ik} S_k^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu}_k) \tag{85}$$

寄与率 γ_{ik} は $\boldsymbol{x}_i$ が潜在状態 k により得られたとする仮説の事後確率 $P(k|\boldsymbol{x}_i)$ であり次式で与えられる。

$$\gamma_{ik} = \frac{w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, S_k)}{\sum_k w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, S_k)} = \frac{P(k, \boldsymbol{x}_i)}{\sum_k P(k, \boldsymbol{x}_i)} \tag{86}$$

式 (85) の偏微分を 0 にする $\boldsymbol{\mu}_k$ を平均の推定値 $\hat{\boldsymbol{\mu}_k}$ として計算する。

$$\hat{\boldsymbol{\mu}_k} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \boldsymbol{x}_i, \quad N_k = \sum_{i=1}^{N} \gamma_{ik} \tag{87}$$

同様にして分散共分散行列の推定値 $\hat{S_k}$ も計算できる。

$$\hat{S_k} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} (\boldsymbol{x}_i - \boldsymbol{\mu}_k)\ {}^t(\boldsymbol{x}_i - \boldsymbol{\mu}_k) \tag{88}$$

重みの推定値 $\hat{w_k}$ は式 (81) 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。

$$\hat{w_k} = \frac{N_k}{N} \tag{89}$$

寄与率 γ_{ik} が求まればパラメータ推定値 $\hat{\theta_k}$ も計算できるが、そもそも γ_{ik} の計算に $\hat{\theta_k}$ が必要なので、**補助関数法**を導入し、目的関数 $\mathcal{L}$ の下限たる補助関数 Q の最大化により間接的に $\mathcal{L}$ の最大化を目指す。

$$\mathcal{L}(\theta) = \max_{\gamma} Q(\gamma, \theta) \tag{90}$$

次式のように γ と θ を交互に修正しつつ反復的に $\mathcal{L}$ を最大化していく。

$$\gamma^{t+1} = \arg\max_{\gamma} Q(\gamma, \theta^t), \tag{91}$$

$$\theta^{t+1} = \arg\max_{\theta} Q(\gamma^{t+1}, \theta) \tag{92}$$

ここで $\mathcal{L}(\theta^t) = Q(\gamma^t, \theta^t)$ と定義すると、式 (91) と式 (92) を交互に繰り返すことで $\mathcal{L}$ は単調増加する。

$$\mathcal{L}(\theta^{t+1}) = Q(\gamma^{t+1}, \theta^{t+1}) \geq Q(\gamma^{t+1}, \theta^t) \geq Q(\gamma^t, \theta^t) = \mathcal{L}(\theta^t) \tag{93}$$

この補助関数法を混合正規分布モデルに適用することを考える。**イェンゼンの不等式**で Q を導出する。ここで $f(x)$ を実数上の凸関数、p_i を $\sum_i p_i = 1$ となる正の実数列とすると、以下の不等式が得られる。

$$\sum_i p_i f(x_i) \geq f\left(\sum_i p_i x_i\right) \tag{94}$$

log が凹関数であることに注意して正負を入れ替えつつ式 (94) を式 (84) に適用することで次式を得る。

$$\mathcal{L}(\theta) \geq \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_{ik} \log \frac{w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, S_k)}{\gamma_{ik}} = Q(\gamma, \theta), \quad \sum_k^K \gamma_{ik} = 1 \tag{95}$$

後は Q を最大化する γ と $\hat{\theta}$ を求めるだけだが、寄与率 γ についてはラグランジュの未定乗数法により式 (86) を、パラメータ推定値 $\hat{\theta}$ については $Q(\theta)$ の偏微分により式 (87)(88)(89) を得る。補助関数を導入せずとも最終的には同じ式が得られたわけであるから、補助関数法の意義がないように思えるかもしれないが、尤度 $\mathcal{L}$ が必ず単調増加して収束解が得られることを保証する上で重要な議論である。

7.3 最尤推定の実装

第 7.2 節で解説したように最尤推定に補助関数法を組み合わせて収束解を得る手法を**期待値最大化法**と呼ぶ。期待値最大化法は式 (86) で γ_{ik} を計算する E ステップと、式 (87)(88)(89) で $\hat{\boldsymbol{\mu}_k}$ と $\hat{S_k}$ と $\hat{w_k}$ を計算する M ステップを反復する。ただし今回は S_k が対角行列であると仮定してベクトルで代用する。

$$\hat{S_{kd}} = \frac{1}{N_k}\left\{\sum_{i=1}^{N}\gamma_{ik}x_{id}^2\right\} - \mu_{kd}^2 \tag{96}$$

では、実装に入ろう。まず EM クラスを定義する。訓練データは 1 次元とし、潜在状態の数を K とする。

```
class EM(samples: Seq[Double], K: Int) {
  val P = Array.ofDim[Double](K, samples.size)
  val W = Array.ofDim[Double](K)
  val M = Array.ofDim[Double](K)
  val S = Array.ofDim[Double](K)
}
```

P は $P(k, \boldsymbol{x}_i|\theta_k)$、W は w_k、M は $\boldsymbol{\mu}_k$、S は S_k を格納する。$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, S_k)$ を求めるメソッドを実装する。

```
  val denom = 1 / Math.sqrt(2 * Math.PI)
  def normal(x: Double, m: Double, s: Double) = {
    val pow = 0.5 * (x - m) * (x - m) / s
    denom / Math.sqrt(s) * Math.exp(- pow)
  }
```

後はコンストラクタで期待値最大化の本体を実装するだけだが、初期化を忘れずに記述しておこう。

```
  for (k <- 0 until K) {
    W(k) = Math.random
    M(k) = Math.random
    S(k) = Math.random
  }
  val wsum = W.sum
  for (k <- 0 until K) W(k) /= wsum
```

初期化を忘れずに記述したら、期待値最大化の本体を記述して EM クラスは完成である。

```
  for (step <- 1 to 100) {
    for ((x, i) <- samples.zipWithIndex) {
      for (k <- 0 until K) P(k)(i) = W(k) * normal(x, M(k), S(k))
      val sum = (for (k <- 0 until K) yield P(k)(i)).sum
      for (k <- 0 until K) P(k)(i) = P(k)(i) / sum
    }
    for (k <- 0 until K) {
      val nk = P(k).sum
      M(k) = (P(k) zip samples).map{case(p, x) => p * x * 1}.sum / nk
      S(k) = (P(k) zip samples).map{case(p, x) => p * x * x}.sum / nk
      S(k) = S(k) - M(k) * M(k)
    }
  }
```

以上で期待値最大化法の実装は完了した。Table 3 のパラメータに従う訓練データを学習させてみた。

Table 3　training data.

w_k	$\boldsymbol{\mu}_k$	S_k
0.57454	0.89739	0.69749
0.42546	0.19543	0.83826

Table 4　training result.

w_k	$\boldsymbol{\mu}_k$	S_k
0.56005	0.81515	0.57402
0.43995	0.22432	0.82382

1,000 件の訓練データに対して 100 回 E・M ステップを繰り返した結果が Table 4 である。収束に至る回数は初期化時の乱数生成に依存するため、掲載プログラムのように回数を決め打ちにするのではなく尤度を式 (84) に従い計算し、尤度が収束した時点で学習を打ち切ると良い。なお、今回は主に紙面の都合により訓練データを 1 次元に限定して実装を単純化したが、Table 3, 4 では推定結果の良し悪しを視覚的に判断しにくい。そこで、多次元空間に対応するように改良を加えて、Fig. 17 に示す散布図を学習させてみた。等高線は 1,000 個の訓練データに対し E・M ステップを 2 回繰り返した結果である。

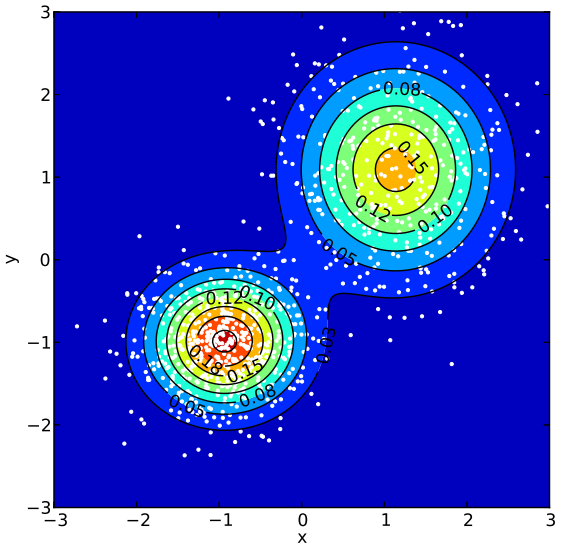

Fig. 17　expectation maximization on gaussian mixture model (2 steps).

学習結果をアニメーション表示して徐々にモデルが改善されていく様子を観察すると面白いだろう。