SVM
(support vector machine)

機械学習演習 プランナークラス

masada@rikkyo.ac.jp

パーセプトロンとSVM

線形モデルによる2値分類

- 確率を使う
  - 線形モデルの出力を確率（０～１の値）に変換
  - 変換にはシグモイド関数を使う

- 別の方法はないか？・・・符号（＋かーか）を使う
  - 実はロジスティック回帰も符号で分けている（後述）

- 符号を直接使うように、損失関数を変更する

パーセプトロン

- 2値分類に使われる機械学習の手法のひとつ
- モデルは線形モデル

$$f(\boldsymbol{x}_i) = a_0 + a_1 x_{i,1} + \cdots + a_d x_{i,d}$$

- 損失関数を以下のように設定する
  - y_iは、i番目のデータ点が属するクラスを**1か−1**で表す、target値。

$$L(a_0, a_1, \ldots, a_d) = \sum_{i=1}^{N} \max(0, -y_i f(\boldsymbol{x}_i))$$

なぜこの損失関数で2値分類できるのか？

- $y = 1$の場合
  - 線形関数の出力$f(\boldsymbol{x})$が正だと、$\max\left(0, -yf(\boldsymbol{x})\right) = 0$
  - 線形関数の出力$f(\boldsymbol{x})$が負だと、$\max\left(0, -yf(\boldsymbol{x})\right) = -f(\boldsymbol{x}) > 0$

- $y = -1$の場合
  - 線形関数の出力$f(\boldsymbol{x})$が正だと、$\max\left(0, -yf(\boldsymbol{x})\right) = f(\boldsymbol{x}) > 0$
  - 線形関数の出力$f(\boldsymbol{x})$が負だと、$\max\left(0, -yf(\boldsymbol{x})\right) = 0$

勾配を使ったパラメータ更新の3種類

- バッチ勾配降下法
  - 一度に訓練データすべてを使って勾配を計算
- ミニバッチ勾配降下法 ← 最もよく使われる
  - 訓練データのうち数十〜数百のサンプルを使って勾配を計算
- SGD（確率的勾配降下法）
  - 訓練データからひとつずつサンプルを使って勾配を計算

SGD (stochastic gradient descent)

- 訓練データのひとつひとつについて勾配を求める、パラメータを更新していく方法

1. パラメータをランダムに初期化
2. 訓練データをランダムシャッフル
3. for each サンプル in 訓練データ:
    1. 一個のサンプルについて勾配を計算
    2. その勾配に小さな数αを掛けて現在のパラメータから引き算する
4. 計算が収束していなければ2. に戻る

パーセプトロンのSGDの更新式

- 実際に偏微分の計算をすると、以下のようになる

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} \leftarrow \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} - \alpha(-y_i) \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

パーセプトロンのSGDの更新式

- $y_i = 1$のとき

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} \leftarrow \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

- $y_i = -1$のとき

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} \leftarrow \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} - \alpha \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

SVM

参考: https://www.ism.ac.jp/~fukumizu/ISM_lecture_2006/svm-ism.pdf

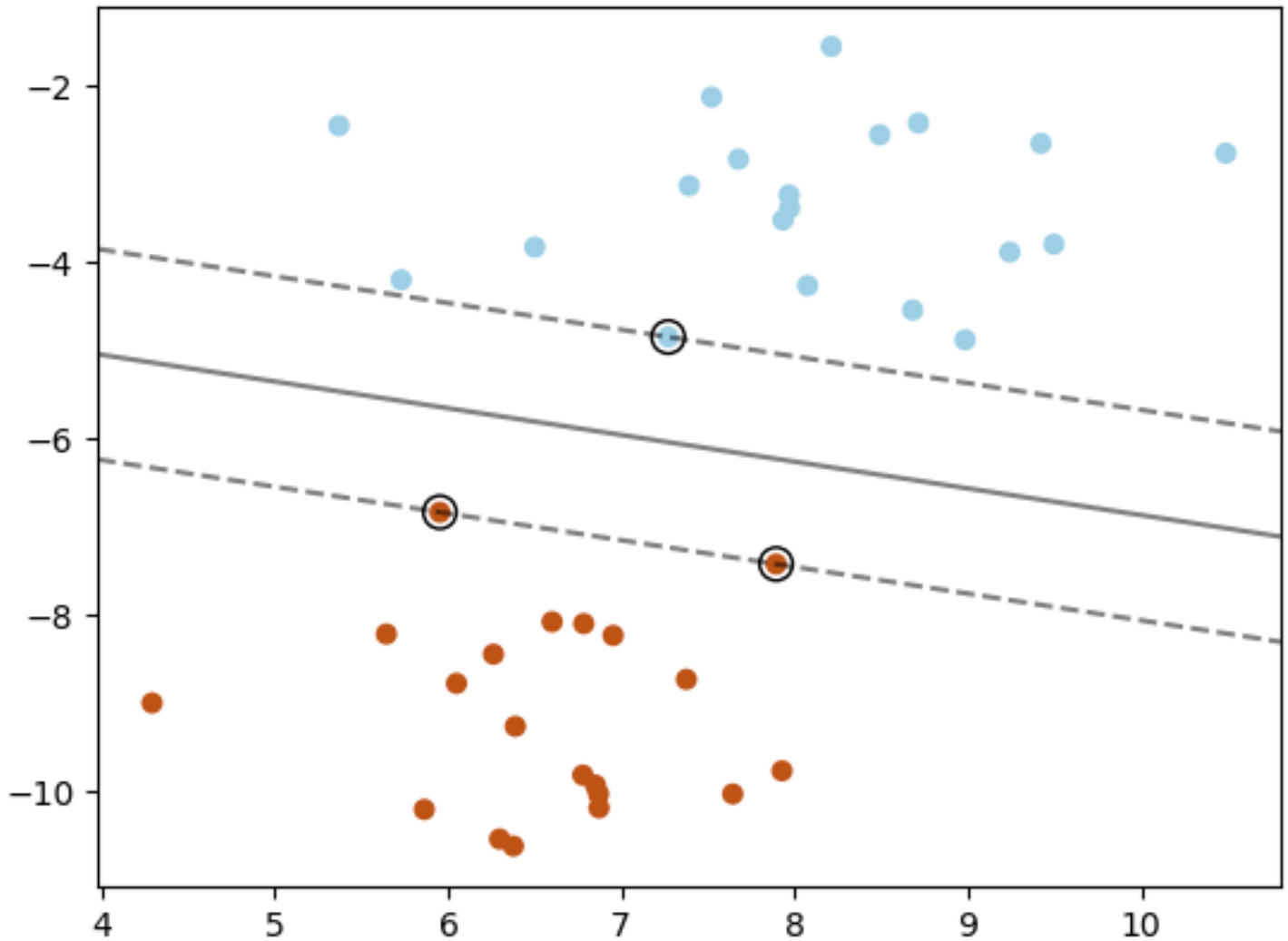
$$f(\boldsymbol{x}_i) = a_0 + a_1 x_{i,1} + \cdots + a_d x_{i,d}$$

- SVMで最小化する関数

$$l(a_0 \ldots, a_d) = \frac{1}{2}\sum_{j=1}^{d} {a_j}^2 + C\sum_i \max(0, 1 - y_i f(\boldsymbol{x}_i))$$

- パーセプトロンで最小化する関数

$$l(a_0 \ldots, a_d) = \sum_i \max(0, -y_i f(\boldsymbol{x}_i))$$



https://scikit-learn.org/stable/modules/svm.html

SVMとパーセプトロンの違い

- $\max(0, 1 - y_i f(\boldsymbol{x}_i))$に注目
  - ヒンジロス(hinge loss)と呼ばれる損失関数
  - $y_i f(\boldsymbol{x}_i)$が1以上にならないと損失がゼロにならない
    - パーセプトロンでは、$y_i f(\boldsymbol{x}_i)$が正なら損失はゼロになっていた。
- 係数の2乗の和も同時に最小化
  - これはマージン最大化に由来する項
    - 一種の正則化ともみなせる

（復習）ロジスティック回帰

- ロジスティック回帰の損失関数はクロスエントロピーだった

$$-\{y_i \log p_i + (1 - y_i) \log(1 - p_i)\}$$

  - 正解y_iが1のとき：$-\log p_i = -\log \frac{1}{1+e^{-f(x_i)}} = \log(1 + e^{-f(x_i)})$
  - 正解y_iが0のとき：$-\log(1 - p_i) = -\log\left(1 - \frac{1}{1+e^{-f(x_i)}}\right) = \log(1 + e^{f(x_i)})$

- target値y_iが**1か−1**であるとき、まとめて次のように書き換えられる

$$\log(1 + e^{-y_i f(x_i)})$$

y_iが1か−1で表される場合に分類器の学習時に最小化する関数

- **SVM**

max(0, 1-t)

$$\frac{1}{2}\sum_{j=1}^{d} {a_j}^2 + C\sum_i \max(0, 1 - y_i f(\boldsymbol{x}_i))$$

- **パーセプトロン**

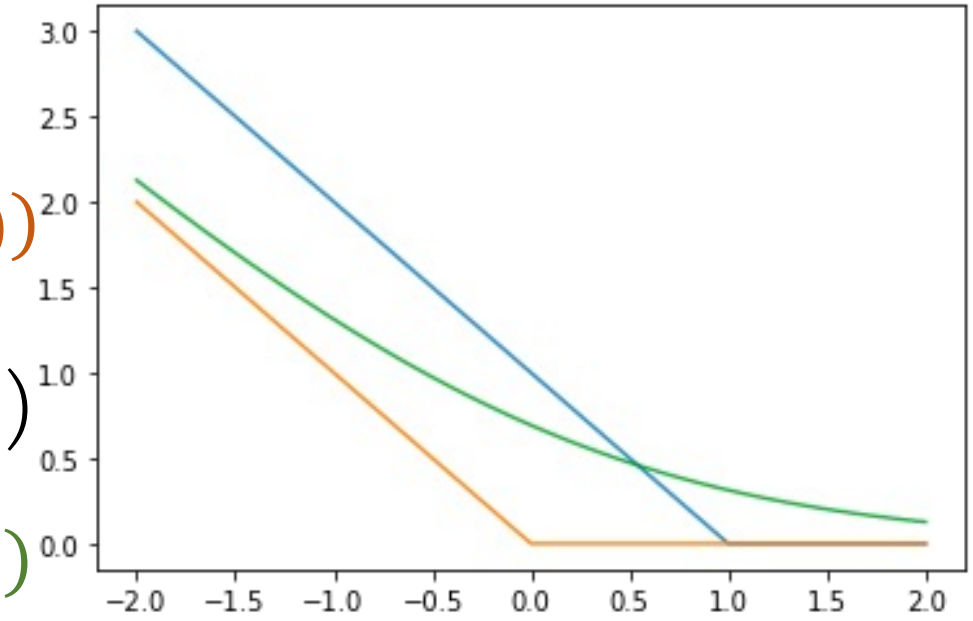
max(0, -t)

$$\sum_i \max(0, -y_i f(\boldsymbol{x}_i))$$

- **ロジスティック回帰**

log(1+exp(-t))

$$\sum_i \log(1 + e^{-y_i f(\boldsymbol{x}_i)})$$



カーネルトリック

パーセプトロンでのパラメータ推定

- パーセプトロンの更新式をよく見ると・・・

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} \leftarrow \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} - \alpha(-y_i) \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

- 推定された$\widehat{\boldsymbol{a}} = (a_1, \ldots, a_d)$は、訓練データ$\boldsymbol{x}_i$の線形結合！

$$\widehat{\boldsymbol{a}} = \sum_{i=1}^{N} w_i y_i \boldsymbol{x}_i$$

  - なお、初期値はゼロベクトルだったとする。

Perceptron Representer Theorem

Theorem 12 (Perceptron Representer Theorem). During a run of the perceptron algorithm, the weight vector is always in the span of the (assumed non-empty) training data.

Hal Daumé III. A Course in Machine Learning. Chapter 11.

http://ciml.info/

パーセプトロンでの予測

- 未知の入力$\boldsymbol{x}' = (x_1', \ldots, x_d')$はどちらのクラス？
  - $\hat{a}_0 + \hat{\boldsymbol{a}}^T \boldsymbol{x}' \geq 0$ならば、$y = 1$のほうのクラス
  - $\hat{a}_0 + \hat{\boldsymbol{a}}^T \boldsymbol{x}' < 0$ならば、$y = -1$のほうのクラス
    - ただし$\hat{a}_0$は推定された切片とする。
- つまり、下の値の符号によってクラスを予測する

$$\hat{a}_0 + \hat{\boldsymbol{a}}^{\boldsymbol{T}} \boldsymbol{x}' = \hat{a}_0 + \left(\sum_{i=1}^{N} w_i y_i \boldsymbol{x}_i \right)^T \boldsymbol{x}' = \hat{a}_0 + \sum_{i=1}^{N} w_i y_i \boldsymbol{x}_i{}^T \boldsymbol{x}'$$

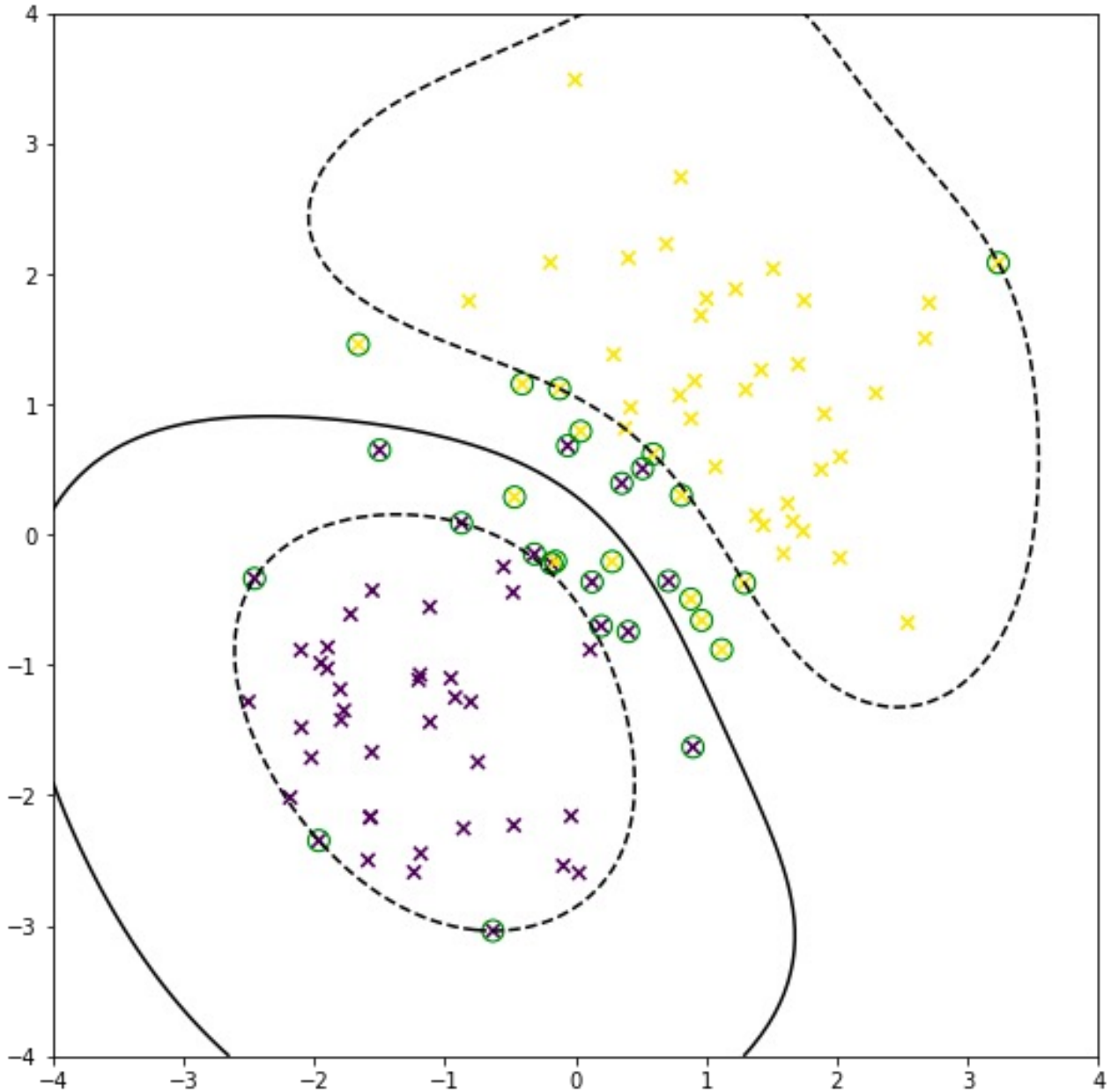
  - 内積を計算できさえすればクラスの予測ができるということ。

高次元化

- 入力ベクトル$\boldsymbol{x} = (x_1, x_2)$を、例えば下の写像$\varphi$で変更してみる

$$\varphi(\boldsymbol{x}) = (\sqrt{2}x_1x_2, {x_1}^2, {x_2}^2)$$

  - 2次の項を含んでいる。
- これを入力ベクトルとして使う
- モデルの表現力が増した
  - 平面ではなく、曲面で分割できるようになる。
    - 別の関数をφとして使った例を、次のスライドに示した。
- でも、必要な計算の量も増えてしまった
  - 次元が上がった空間で内積の計算をしないといけないため。



https://nbviewer.jupyter.org/github/ctgk/PRML/blob/master/notebooks/ch07_Sparse_Kernel_Machines.ipynb

上を単独で動かせるようにしたもの: https://github.com/tomonari-masada/course2022-sml/blob/main/RBF_kernel_example01.ipynb

写像φのうまい選び方

- 予測に必要なのは、入力ベクトルどうしの内積だけだった
- $\varphi(\boldsymbol{x}) = (\sqrt{2}x_1x_2, {x_1}^2, {x_2}^2)$で高次元化したら内積はどうなる？

$$\varphi(\boldsymbol{x})\varphi(\boldsymbol{x}') = \left(\sum_{j=1}^{d} x_j {x_j}'\right)^2 = (\boldsymbol{x}^T\boldsymbol{x}')^2$$

  - 計算の手間が元の入力ベクトルどうしの内積とほぼ同じで済む
- 写像φをうまく決めると、計算の手間は増やさずに、モデルの表現力だけを上げることができる

内積の便利さ

- ベクトルに関する処理には、内積さえ計算できれば実行できるものが結構ある
  - ベクトルの長さを求める（＝自分自身との内積のルート）
  - ベクトル間の距離を求める（＝引き算して長さを求める）
  - 複数のベクトルの重心（平均）から別のベクトルまでの距離
    - 複数のベクトルの重心そのものは、内積だけでは求められない。
  - 主成分分析

カーネル

- 入力ベクトルを写像φで移した先の空間での内積を与える関数を、カーネル関数と呼ぶ
  - $\varphi(\boldsymbol{x}) = (\sqrt{2}x_1x_2, {x_1}^2, {x_2}^2)$ のカーネル関数kは以下の通り。

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}')^2$$

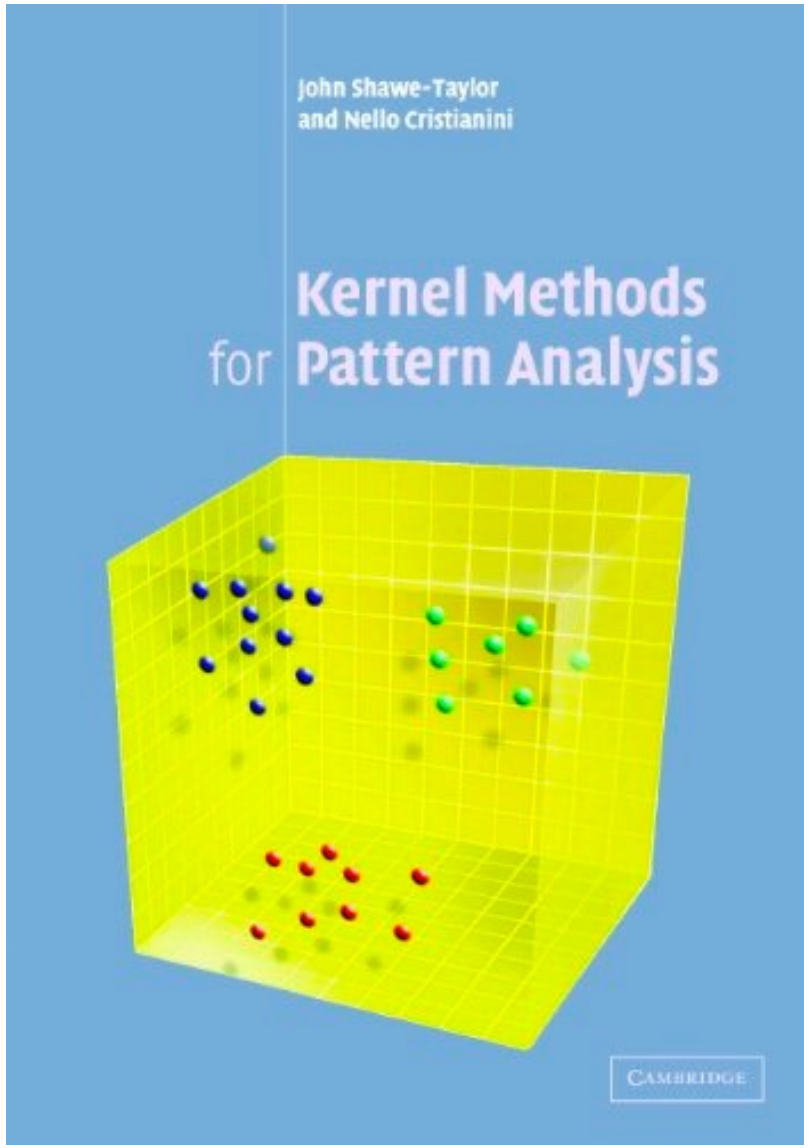
  - 3つ前のスライドのカーネル関数（RBFカーネルと呼ばれる）は、以下。

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \sum_{j=1}^{d} (x_j - {x_j}')^2)$$

- 参考: http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf

カーネル・トリック

- 写像φを、feature mapと呼ぶ
- うまく写像を選ぶと、高次元化した後の空間での内積が、元の空間での内積と同じぐらいの手間で計算できる
  - feature mapが明示的に計算される必要はない
- カーネル・トリック
  - 入力ベクトルをfeature mapで高次元空間に移す
  - 移した先の空間(feature space)で、SVMや主成分分析をおこなう
  - feature spaceでの超平面は、元の空間では曲面になっている
    - より柔軟な分類や次元圧縮がおこなえる。



https://kernelmethods.blogs.bristol.ac.uk/