

データサイエンス実践 「機械学習」

末廣 大貴, Diego Thomas, 正井 克俊

みなさんにとって学習とは？

- ドリルの繰り返し, 模試
 - ◆ いろいろなパターンの問題を解くことで数値や文章が変わっても解ける
- 試験前一夜漬けで答えを記憶！
 - ◆ あまり勧められませんよね
 - ◆ 「答えを記憶する」のは一般に学習するとは言われません
- 大事なのは練習や経験を活かして汎用的な能力とし、
まだ見ぬ本番で発揮するチカラ
 - ◆ 「汎化能力」などと呼ばれます

人間は汎化能力が高い

- 様々なコトから汎用的な知識を学べる
 - ◆ ドリルや模試をたくさん解く→本番の試験でも良い点がとれる



- ◆ いろいろな犬や猫を知る→ この動物は犬？猫？



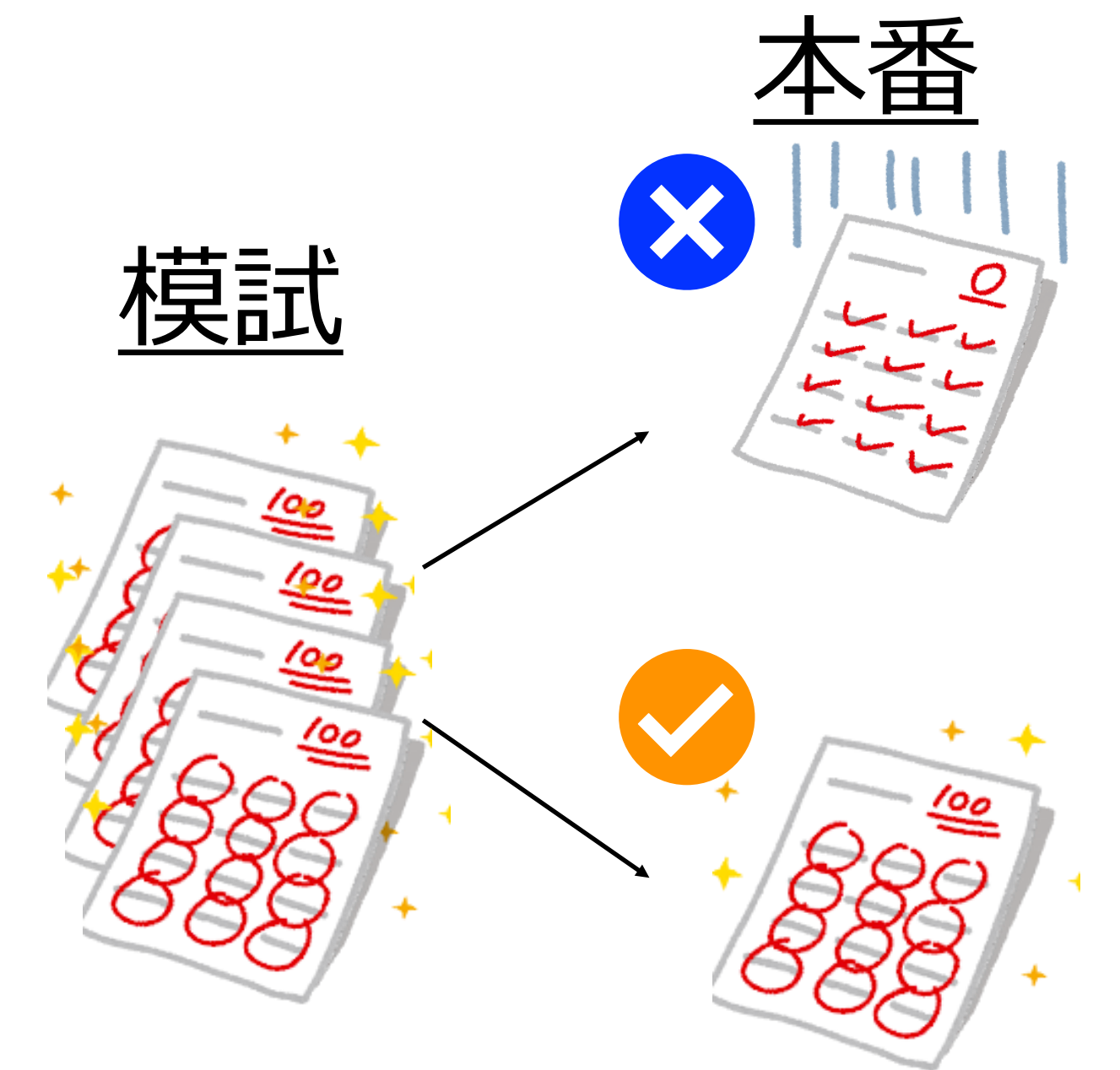
ノルウェイジアン・パフィン・ドッグ
wikipediaより

- ◆ いろんな美味しいオニギリを食べる
→オニギリに入れるとしたらしょっぱいもの？甘いもの？



機械学習に求められるのも汎化能力

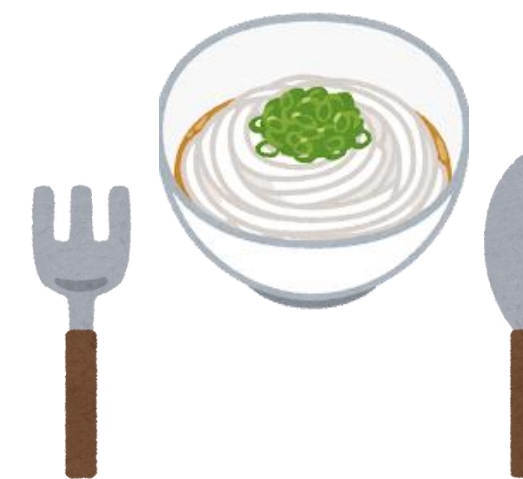
- みなさんの「学習」と同じ
- 模試やドリルの答えを記憶するのは（ほぼ）無意味
 - ◆ 機械学習においても、データの答えを記憶できたところで汎化能力が高いとは言い難い
- 機械学習では学習データを使ってどれくらい「本番」に強い予測モデルを作れるかが重要視されます
 - ◆ 学習データで100%正解できても、テストデータ（本番）で50%しか正解できなければ性能が良いとは言えない
 - ◆ 学習データに対する性能（訓練性能）とテストデータに対する性能（汎化性能）は必ずしも比例しないのが機械学習の「難しい」ところ
 - ◆ 詳細はのちほど



様々な機械学習

良い学習器って？

- 訓練性能と汎化性能のギャップが小さい（当たり前！）
 - ◆ 模試で80点くらいなら本番も80点くらいの性能が出ると良い
- 効率的（計算時間が短い，必要なメモリが少ない）
- その他：色々なデータに使える（実数値，離散値，グラフ，...），実装が簡単，結果が解釈しやすい，などなど
- 残念ながら**無敵の学習器は存在していない**
 - ◆ ひとえに学習と言っても，問題設定は様々
 - ◆ 学習器も適材適所
 - それぞれ得意不得意がある
 - 高コストであれば良いとは限らない
 - 目的に応じて使い分けよう



目的に合わない



オーバースペック

一番シンプルな2クラス分類から

例：身長，体重に関するデータをもとに
健康状態（健康 or 不健康）を予測したい

	身長	体重	健康状態
A さん	175cm	54kg	good
B さん	160cm	70kg	bad
C さん	190cm	100kg	good
D さん	170cm	45kg	bad
⋮	⋮	⋮	⋮

各データの特徴量（身長，体重）を用いて
クラス（健康状態）の予測を算出する問題

ベクトルで表現

- データをベクトル表現に
- クラスラベル（健康状態）を +1, -1 の 2 値表現に

	身長	体重	y
\mathbf{x}_1	175	54	+1
\mathbf{x}_2	160	70	-1
\mathbf{x}_3	190	100	+1
\mathbf{x}_4	170	45	-1
\vdots	\vdots	\vdots	\vdots

おさらい：ベクトルと内積

- ベクトルとは大雑把に言うと「数値の組」
 - ◆ ただし、要素の順番に意味がある
- 身長、体重は2つの数値の組
 - ◆ 数値の組の1番目は身長、2番目は体重、とすると、身長 = 180cm, 体重 = 70kg の人は (180, 70) というベクトルで表現できる
- 内積とは大雑把に言うとベクトル同士の「類似度」
 - ◆ 2つのベクトルが似ていると内積 \asymp 類似度は大
 - ◆ 2つのベクトルが似ていないと内積 \asymp 類似度は小

おさらい：類似度

いわゆる直線距離

- 類似度といえはいわゆるユークリッド距離では？

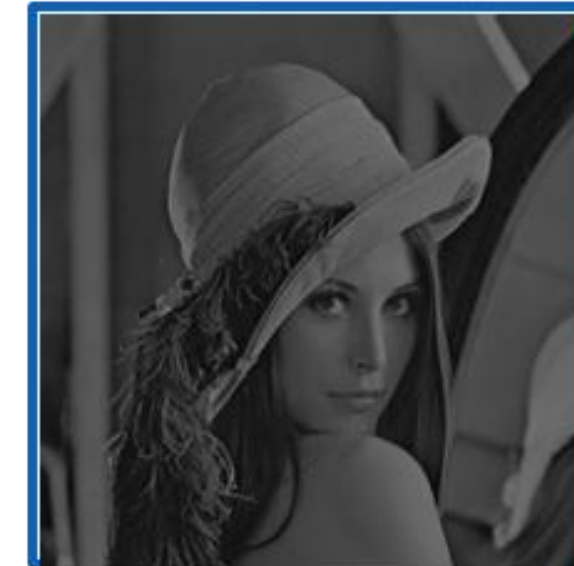
例) 256x256 の画像： 65536次元ベクトル



x_1



x_2



x_3

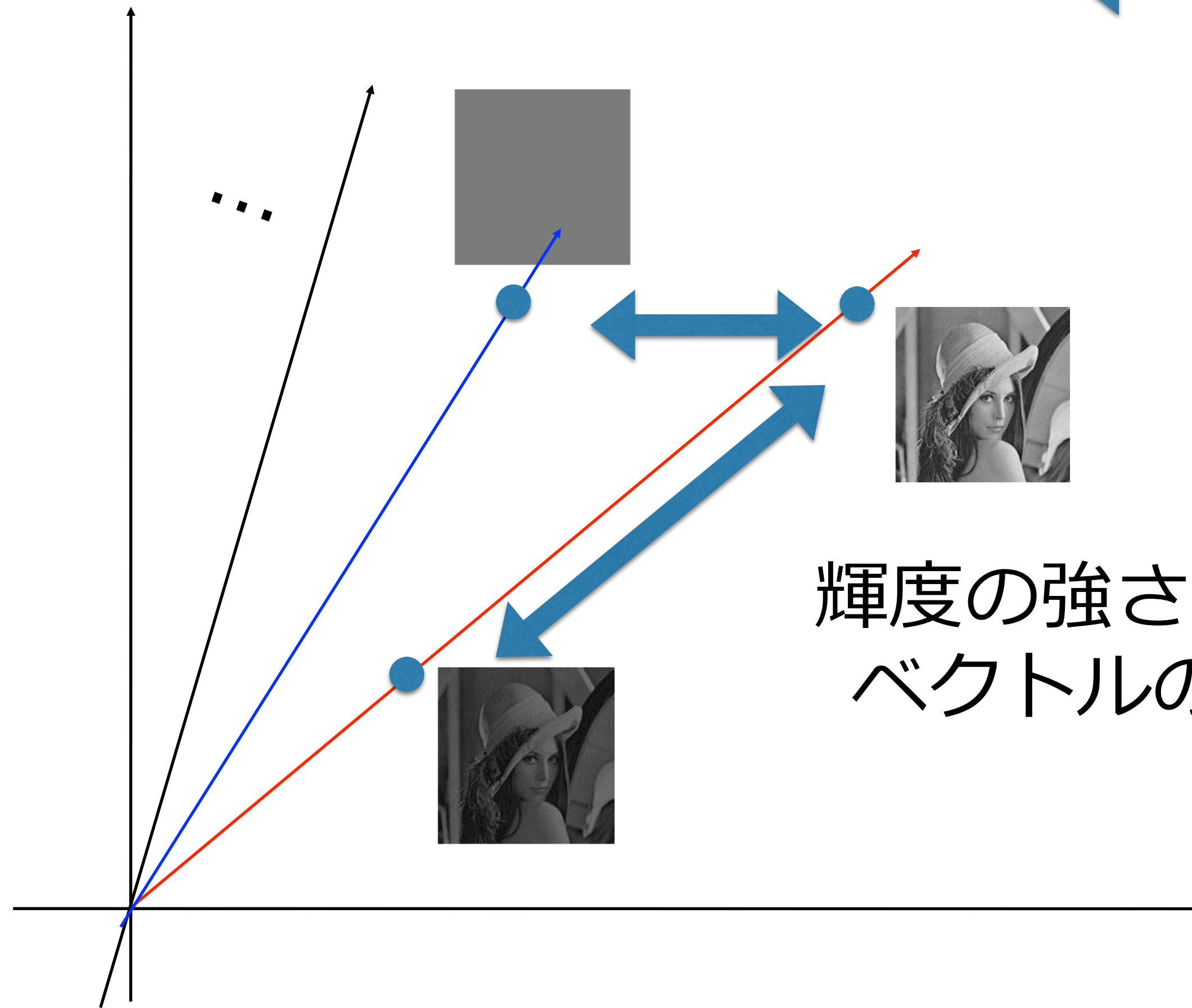
$$\|x_1 - x_2\| = 12249.03$$

$$\|x_2 - x_3\| = 16957.99$$

人の直感とは異なる結果

おさらい：類似度

↔ : ユークリッド距離

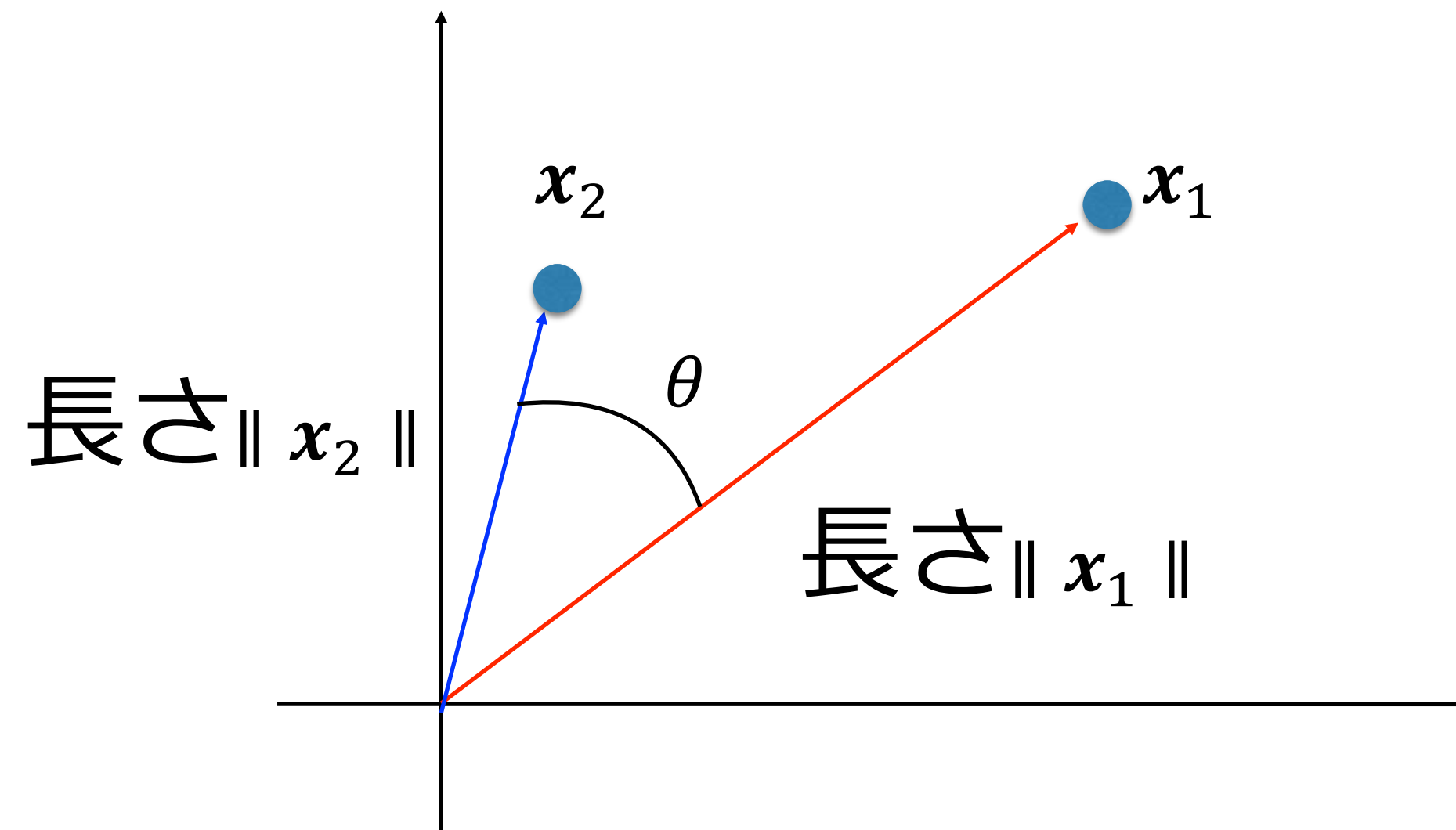


おさらい：内積

- 内積は角度（向き）を考慮した類似度

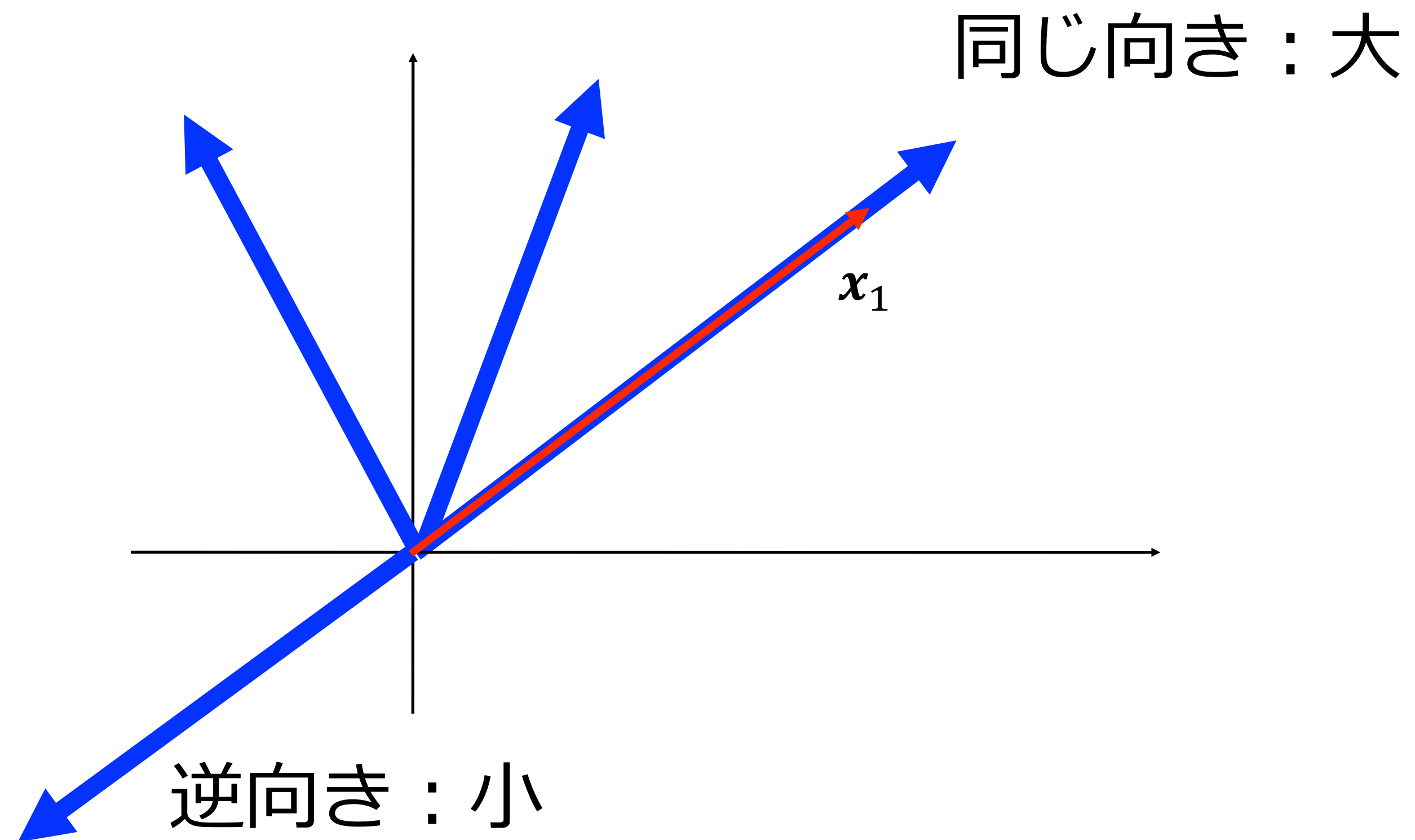
- ◆ $x_1 \cdot x_2 = \|x_1\| \|x_2\| \cos\theta$

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots}$$



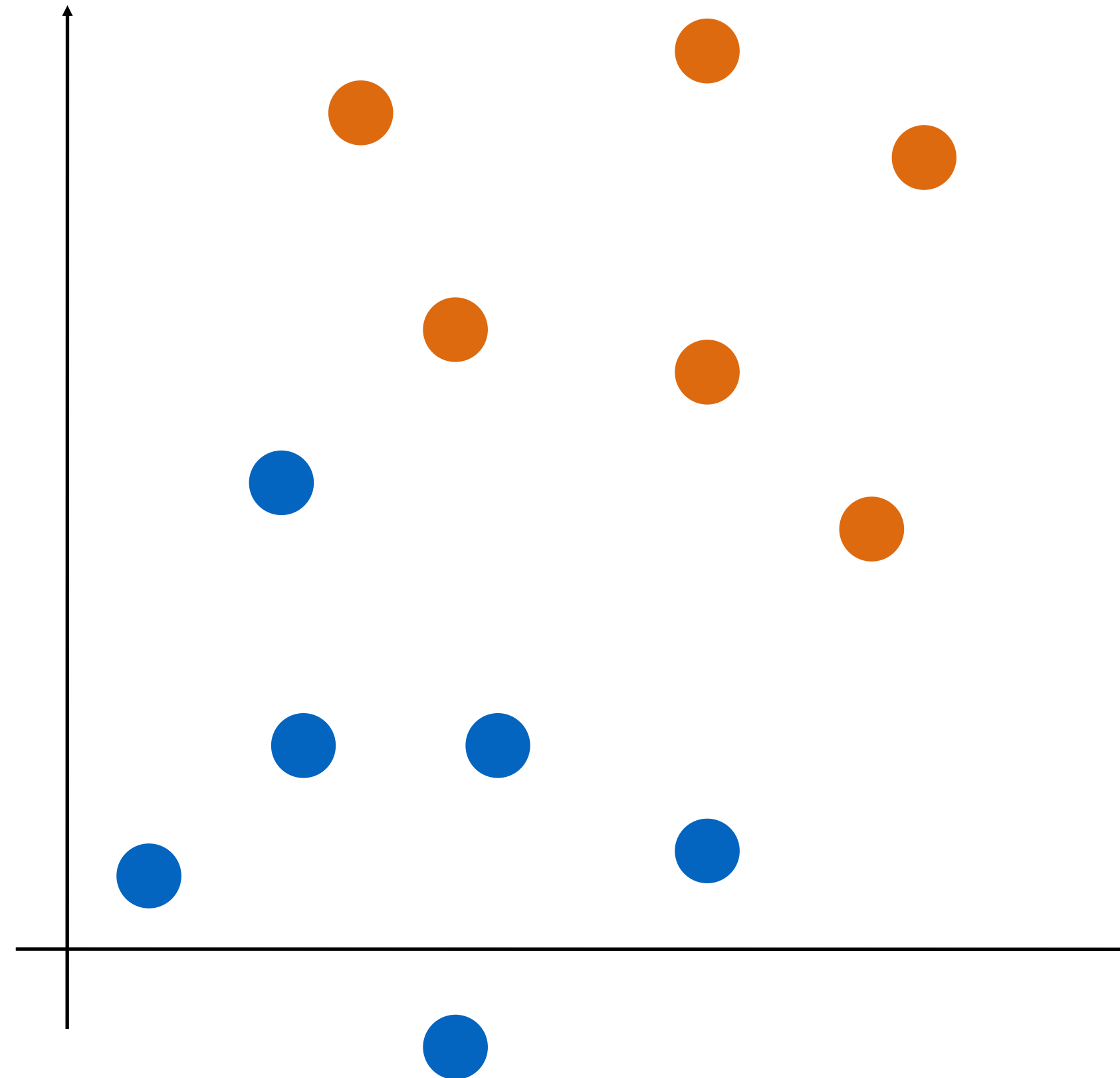
おさらい：内積

- 内積が大きいとき, 小さい時



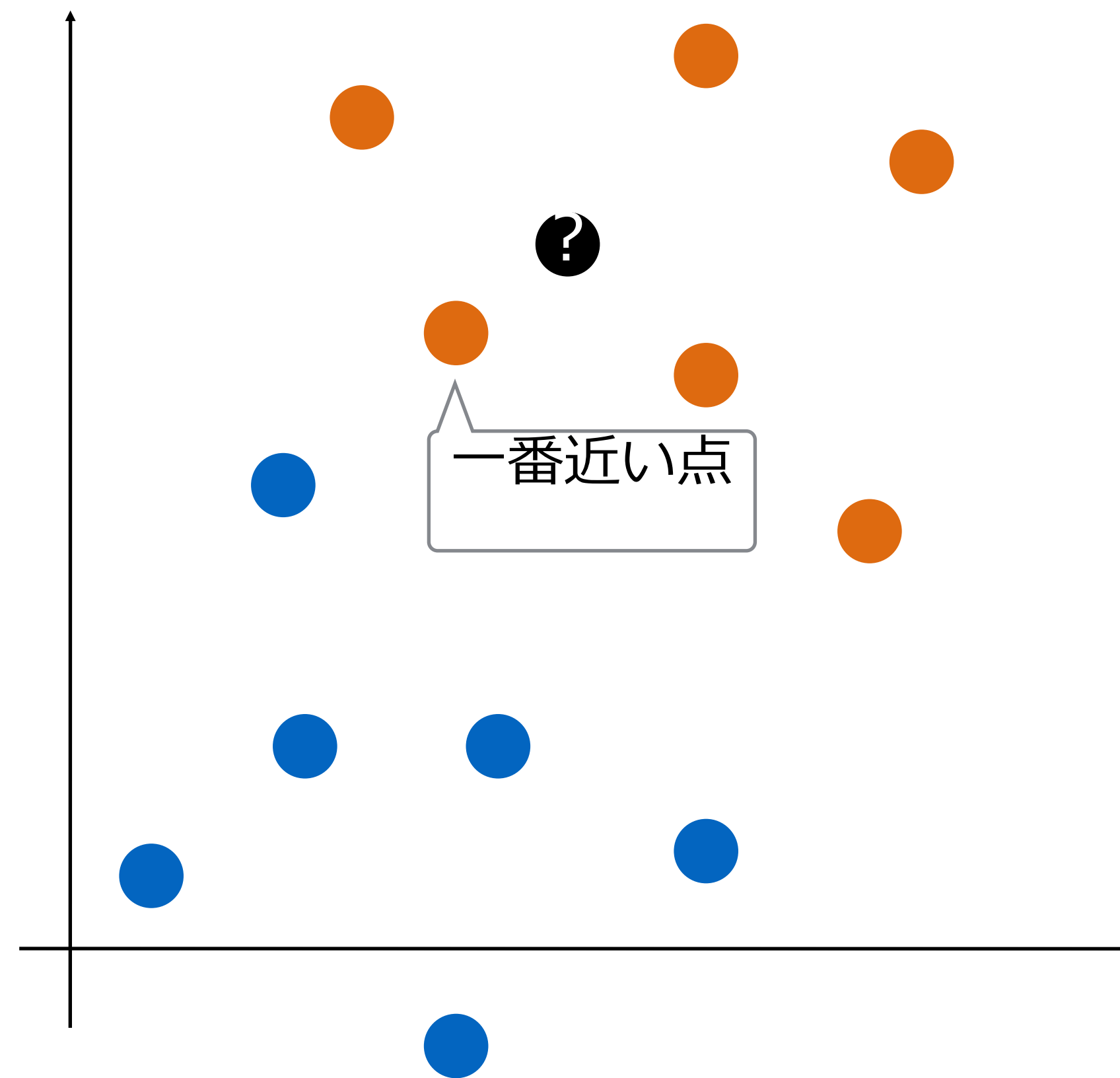
ベクトルを分類するには？

2次元ベクトル
→2個の数値の組
→2次元座標上の点



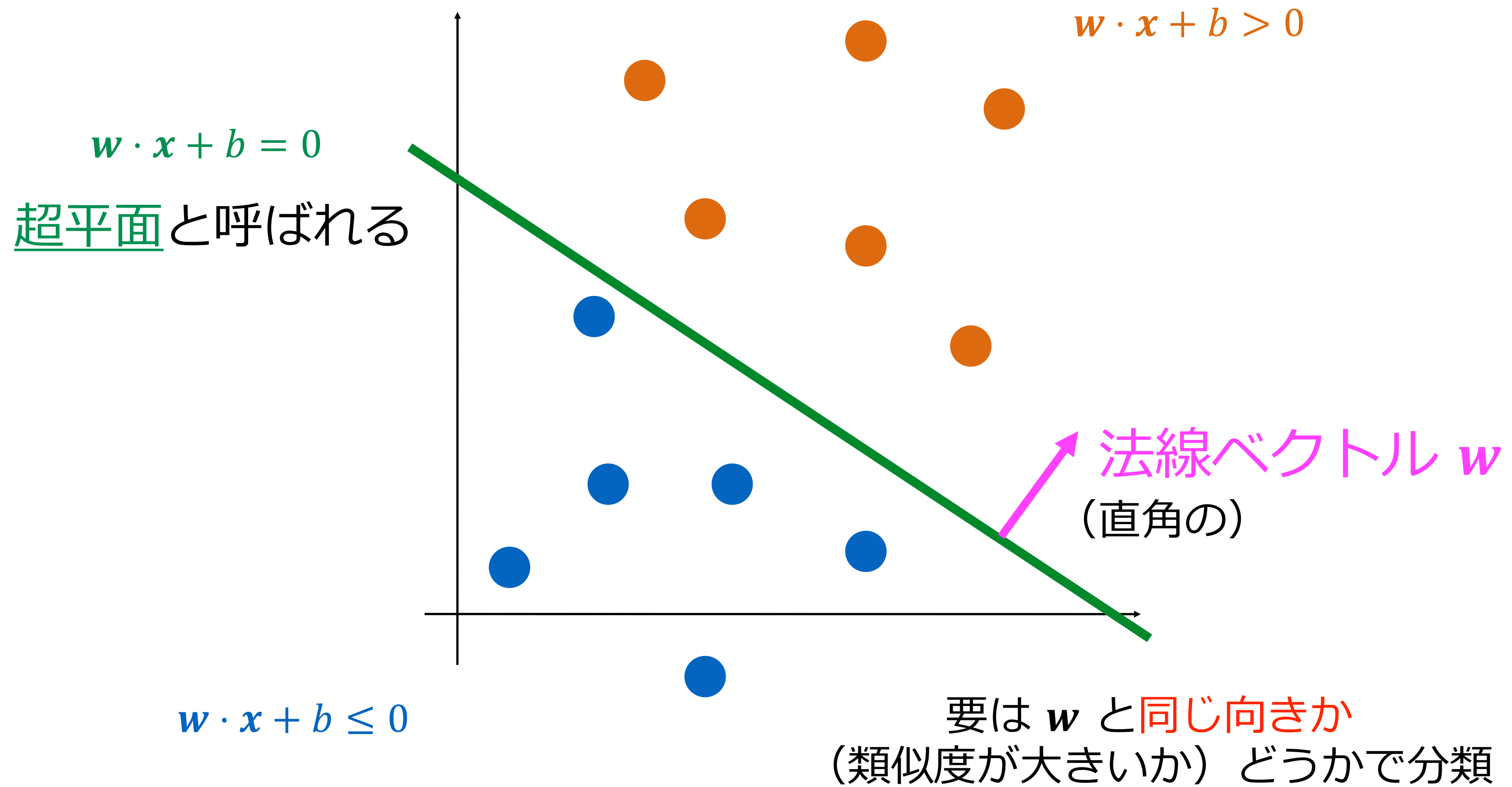
近傍法

最もシンプルな分類方法（学習不要！）

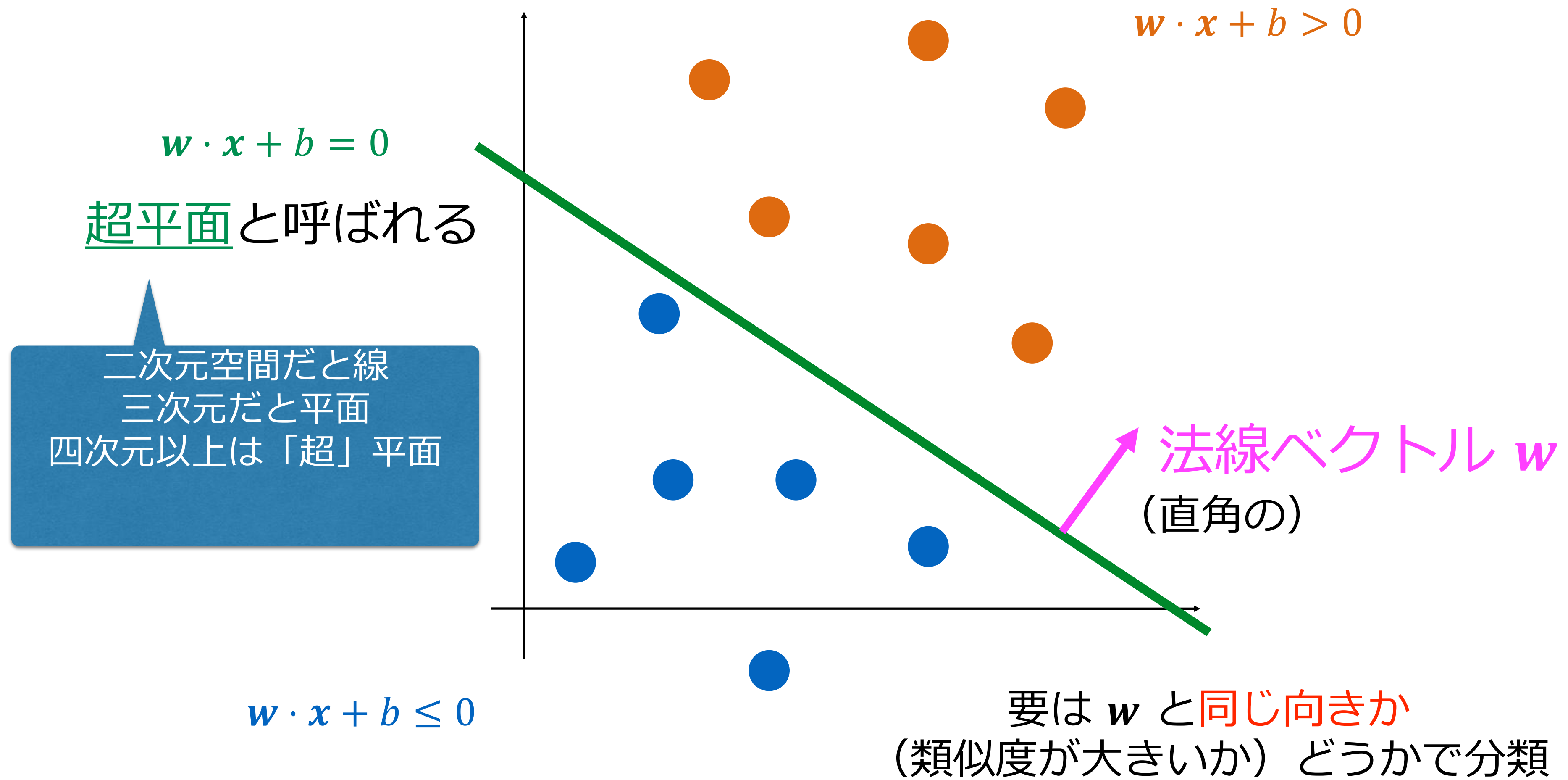


答えが不明なデータが来たら
一番近い点と同じ答えを予測
(この場合一番近い点はオレンジ
なのでオレンジと予測)

線形分類器



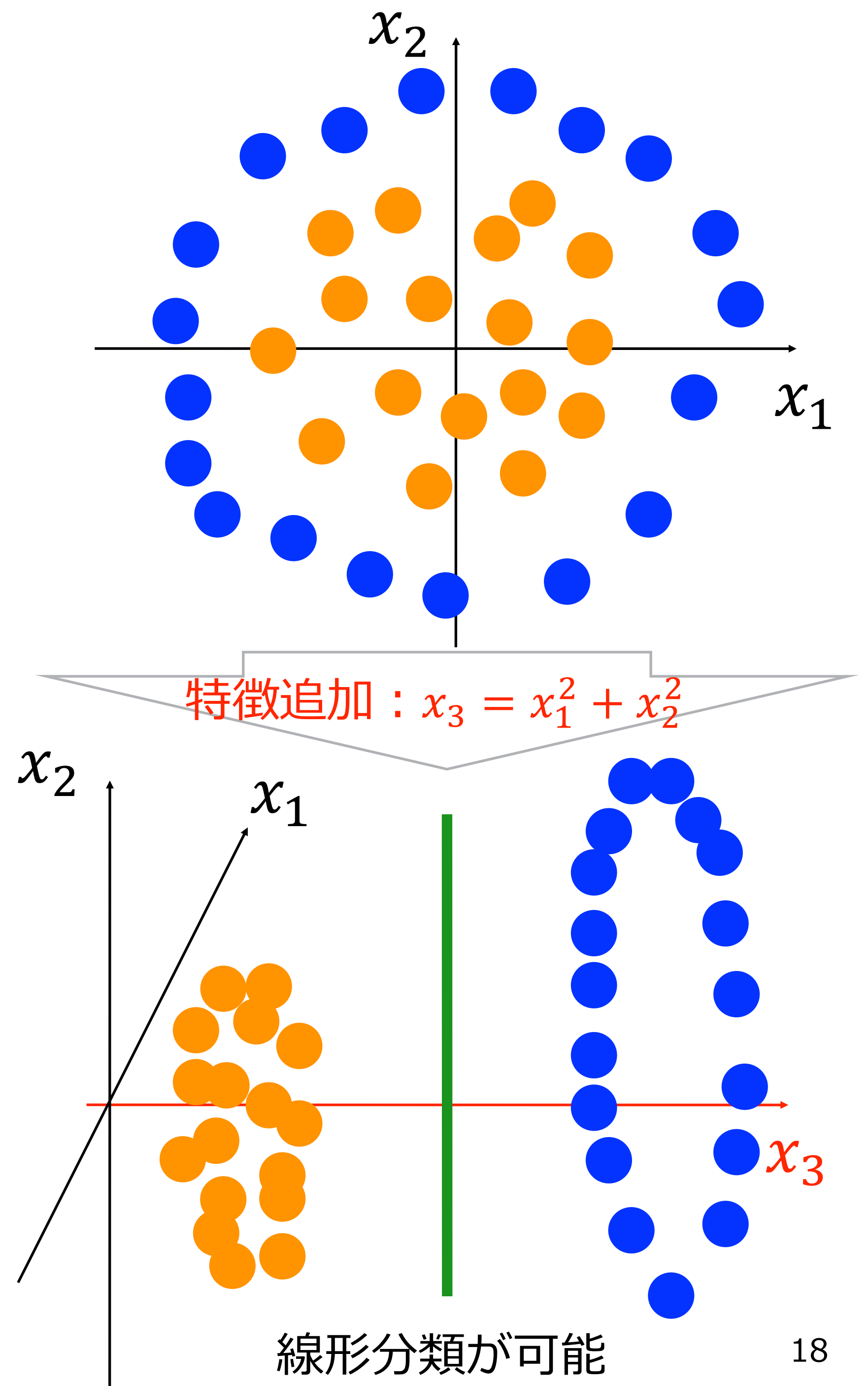
線形分類器



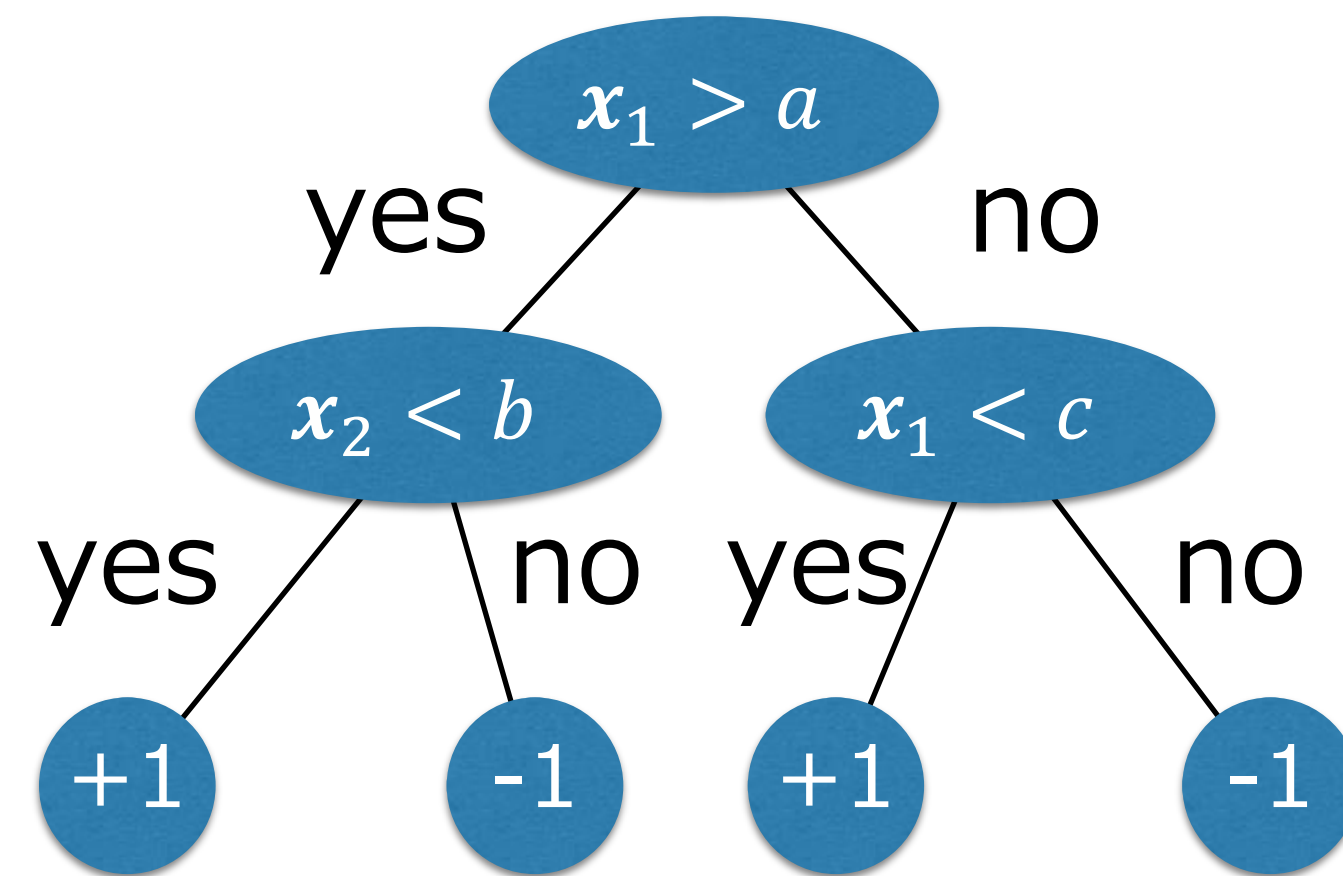
データから w を学習する

非線形分類器

- まっすぐな線でしか分けられないのか？
 - ◆ 非線形（曲線）での分類への拡張は実は簡単
- 非線形分類は
 - ◆ もともとの特徴から**新しい特徴を追加することで可能**になる
 - ◆ 特徴を増やした世界での線形分類がもともとの特徴での非線形分類に対応
 - ◆ 近年話題の深層学習はこの「追加特徴」を学習して作り出すので複雑なことも可能に
- x から特徴追加したものを $\phi(x)$ とすると非線形分類器は $w \cdot \phi(x) + b$ と表現できる



決定木



離散値（5段階アンケート等）や
カテゴリカル変数（あるなし，出身地など）
があっても扱いやすい

例：「まあまあ良かった」以上か？で yes or no
「福岡出身か？」で yes or no

質問を繰り返して結果を得る

どのように木を構築するか？が学習

分類以外でも基本は同じ

■ ランキング

- ◆ 線形 or 非線形： $w \cdot x + b$ ($w \cdot \phi(x) + b$) がスコア付け関数
(ランクが高いほど高いスコアを与えるよう学習)
- ◆ 決定木：葉がスコア（もしくはランク）を返す

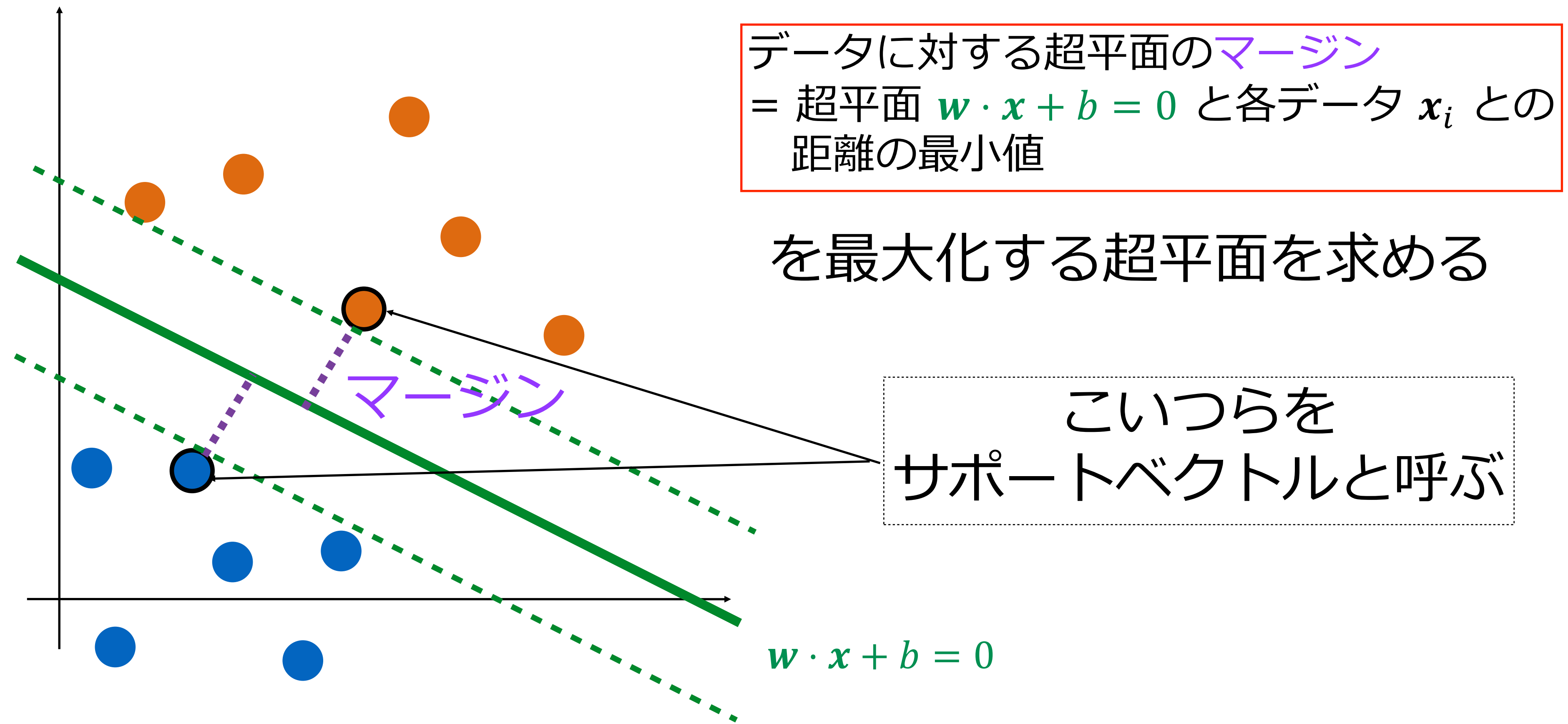
■ 回帰

- ◆ 線形 or 非線形： $w \cdot x + b$ ($w \cdot \phi(x) + b$) が値を与える
- ◆ 決定木：葉がスコア（もしくはランク）値を返す

分類モデルの学習方法

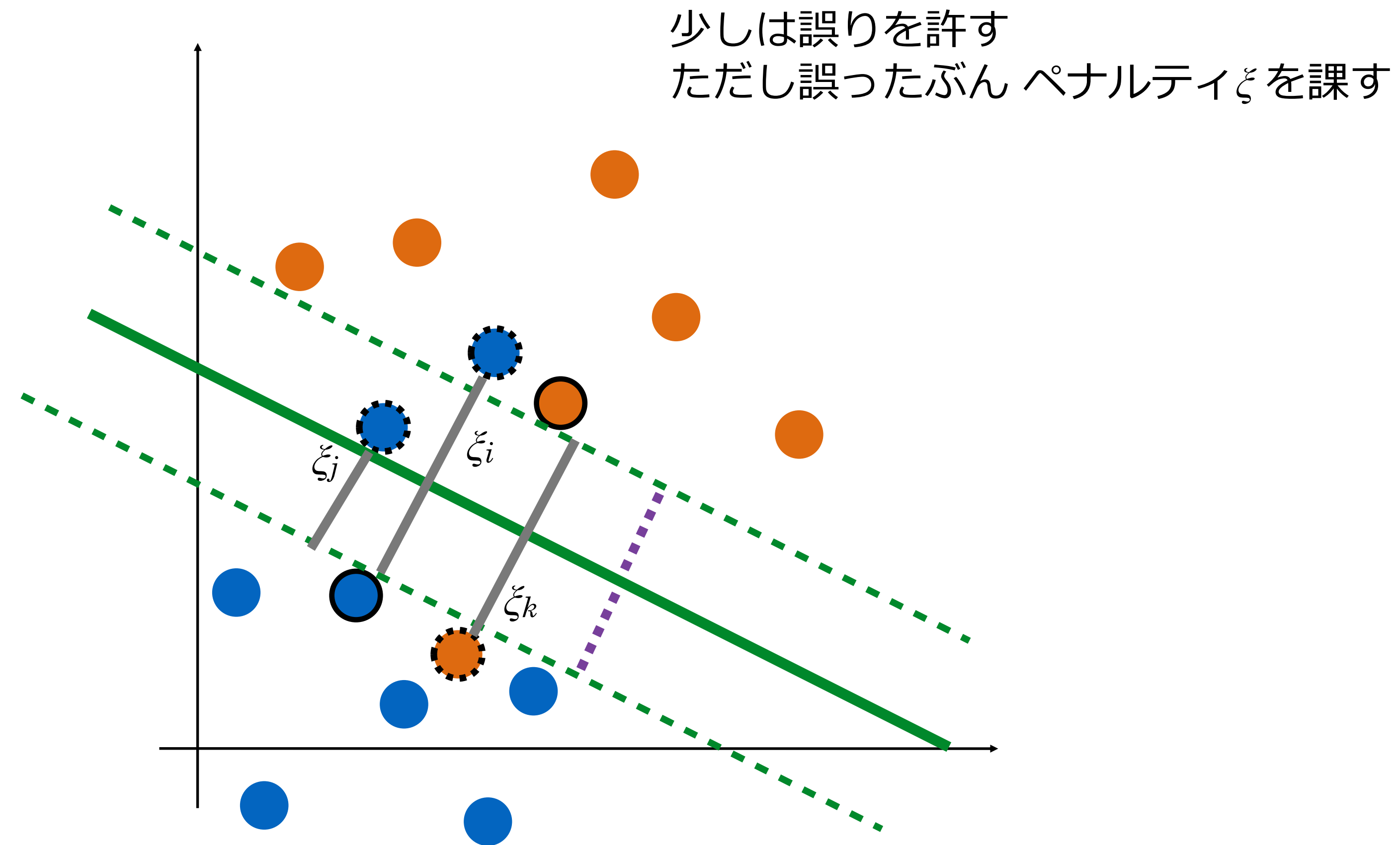
Support Vector Machine (SVM)

[Cortes & Vapnik, 1995]



要するに「余裕を持った境界」を学習する方法

線形分類器で分けられないデータの場合はどうなる？



なるべくマージンを大きくするために、ある程度の誤りは許容する

SVMの「中身」

なるべくマージンを大きくしてね

こんな「最適化問題」を解いています

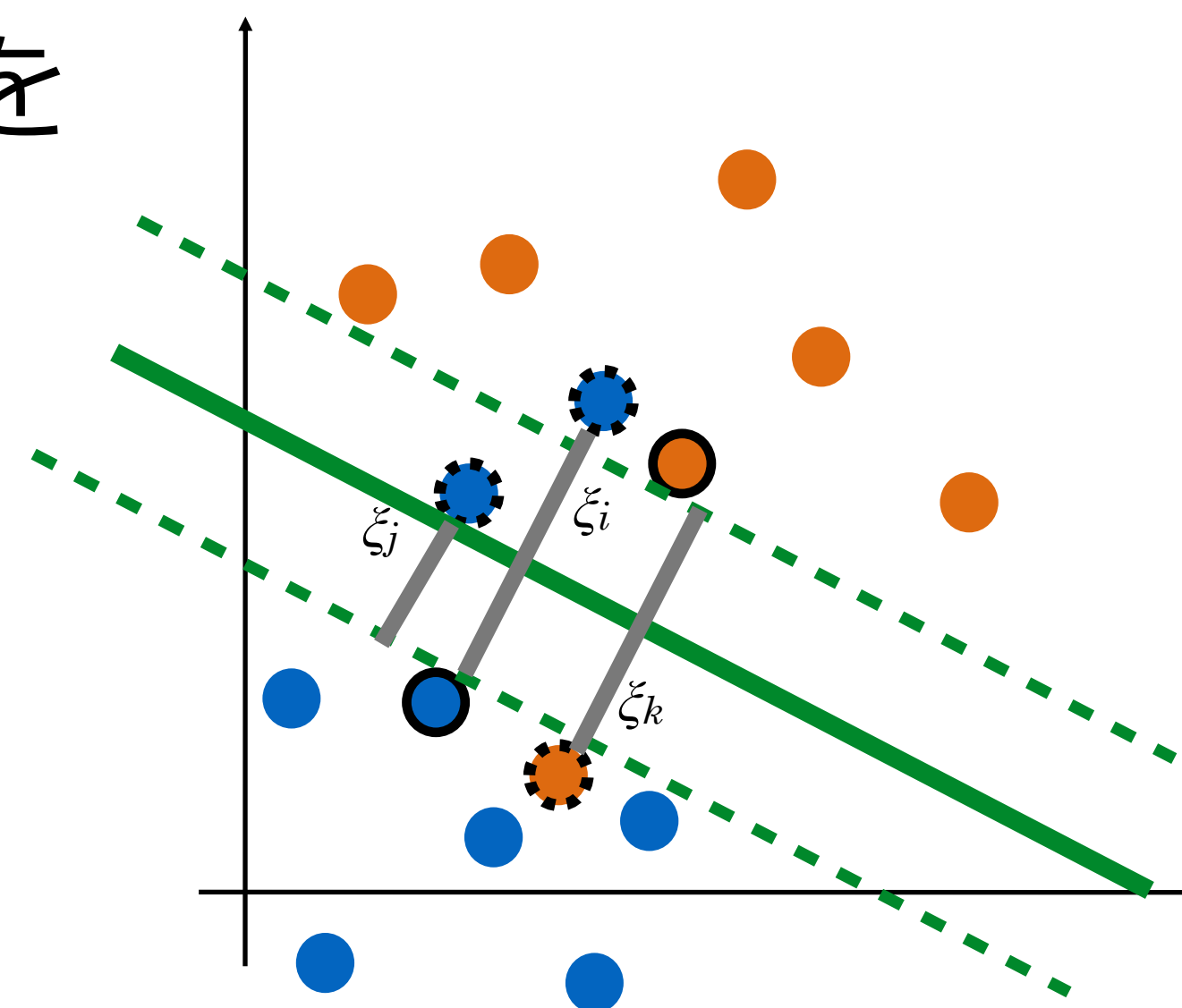
$$\min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^m \xi_i$$

なるべく間違いは少なくしてね

$$\text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \xi_i > 0, i = 1, \dots, m$$

- C はマージン最大化と間違いに伴うペナルティ最小化を制御する「ハイパーパラメータ」

人間が事前に設定する必要のあるパラメータのことをハイパーパラメータといいます



補足：正則化項

SVM以外でも
頻出です

- SVMでは $\min \| \mathbf{w} \|^2$ はマージン最大化の意味がある
 - ◆ $\|\cdot\|$ は「ノルム」と呼ばれ一般的なのは「2ノルム」 ($\| \mathbf{w} \| = \sqrt{w_1^2 + w_2^2 + \dots}$)
 - ◆ 機械学習分野では、ノルムを小さくすることを「正則化」と呼ぶ
 - ◆ 正則化はあらゆる機械学習技術で導入される概念で過学習を抑制できる効果がある
 - ニューラルネットなどでもよく用いられる
- 実装のオプションなどでもよく出てくるので、過学習気味の結果が出ている場合は正則化の効果を強めると改善されることも

SVMのメリット・デメリット

■ メリット

- ◆ 計算コストがそこまで大きくない
- ◆ 比較的データ数が少なくても性能が良い
 - 訓練性能と汎化性能のギャップが小さい
- ◆ 解釈性が高い
 - 線形分類器は得られた重みベクトル w を観察することで各説明変数（特徴量）の寄与度がわかる
- ◆ 様々な拡張が知られている
 - スパース化（得られる w がなるべく0の要素を多く持つように学習）
 - 対高次元データ, データ数が多い場合の高速化
 - マルチクラス, ランキング

■ デメリット

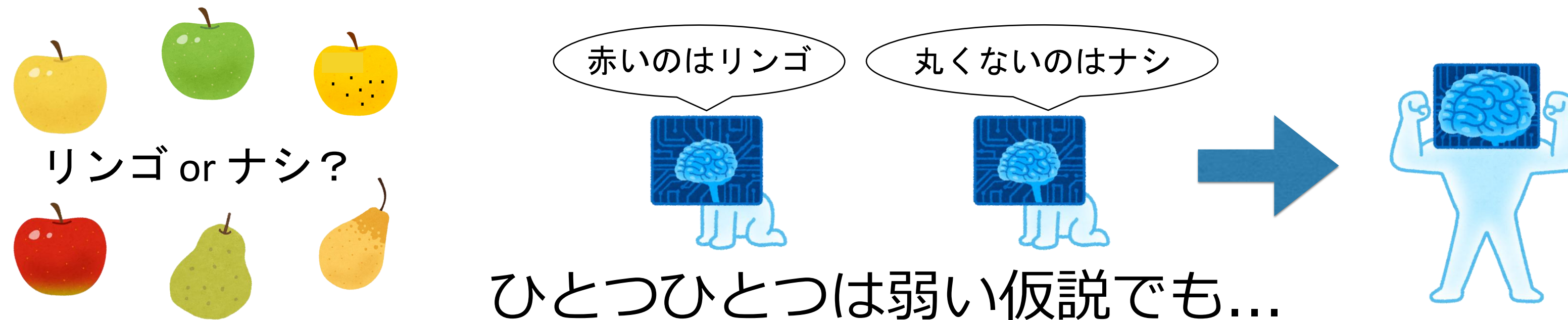
- ◆ 線形分類器は複雑なデータの場合訓練性能が低め
- ◆ 非線形分類器も学習可能（カーネルSVM. 今回は説明省略）だがDNNのような「特徴の学習」はできない
- ◆ 説明変数に離散値が入ると扱いづらい

Random Forest, Boosting (アンサンブル学習)

アンサンブル学習とは？



- ensemble = 集団
- アンサンブル学習：複数の（弱い）仮説を組み合わせるともっと性能の良い（強い）仮説を得ること
- 例：リンゴとナシの分類問題



組み合わせると強い仮説を生成しよう！

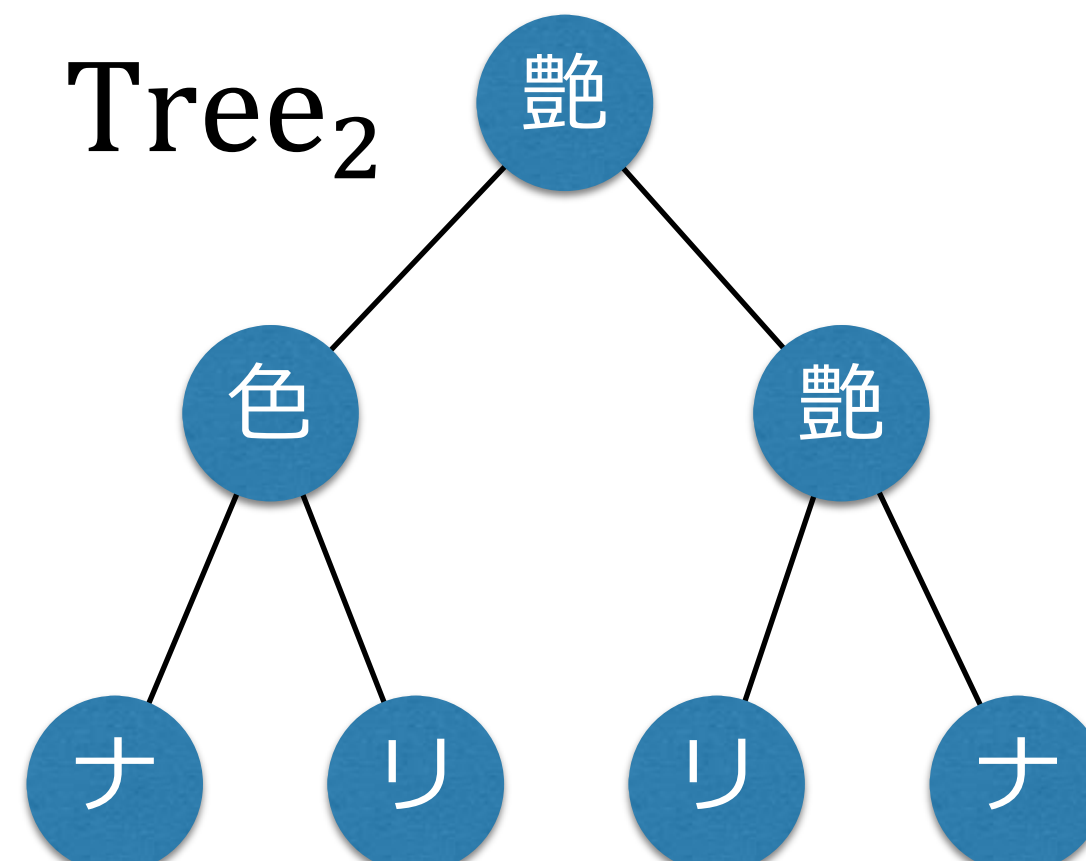
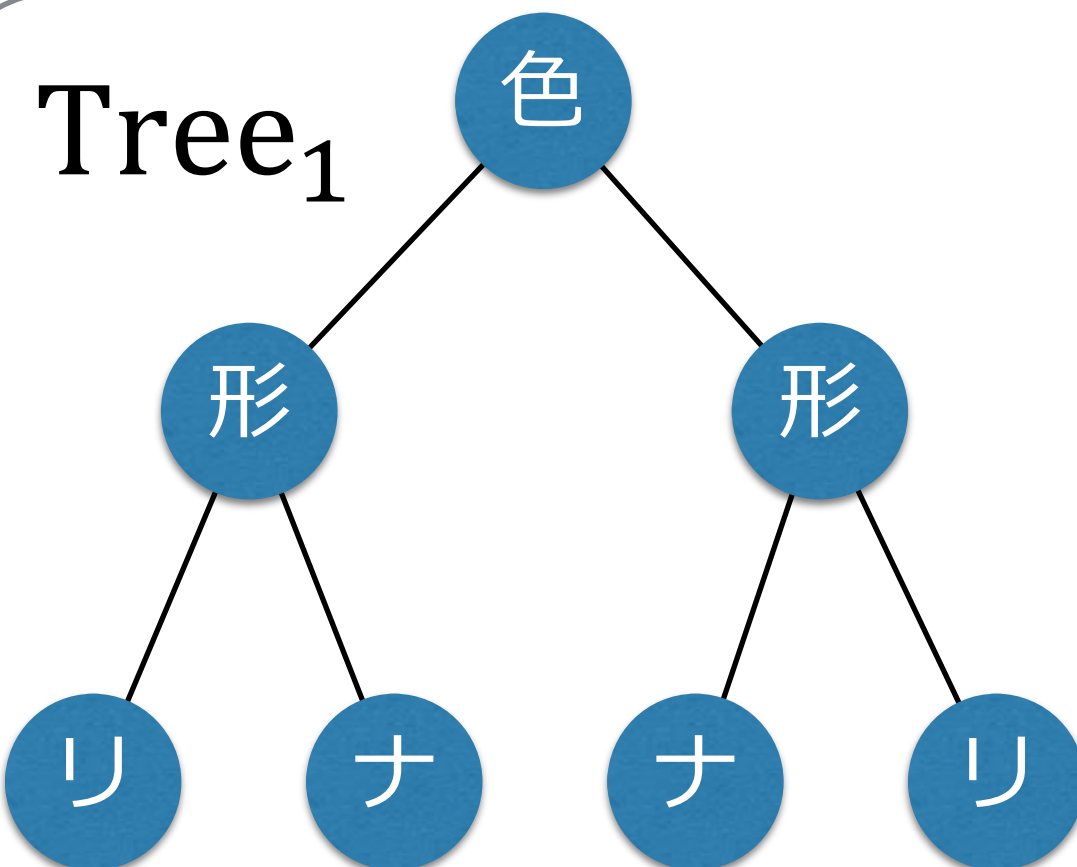
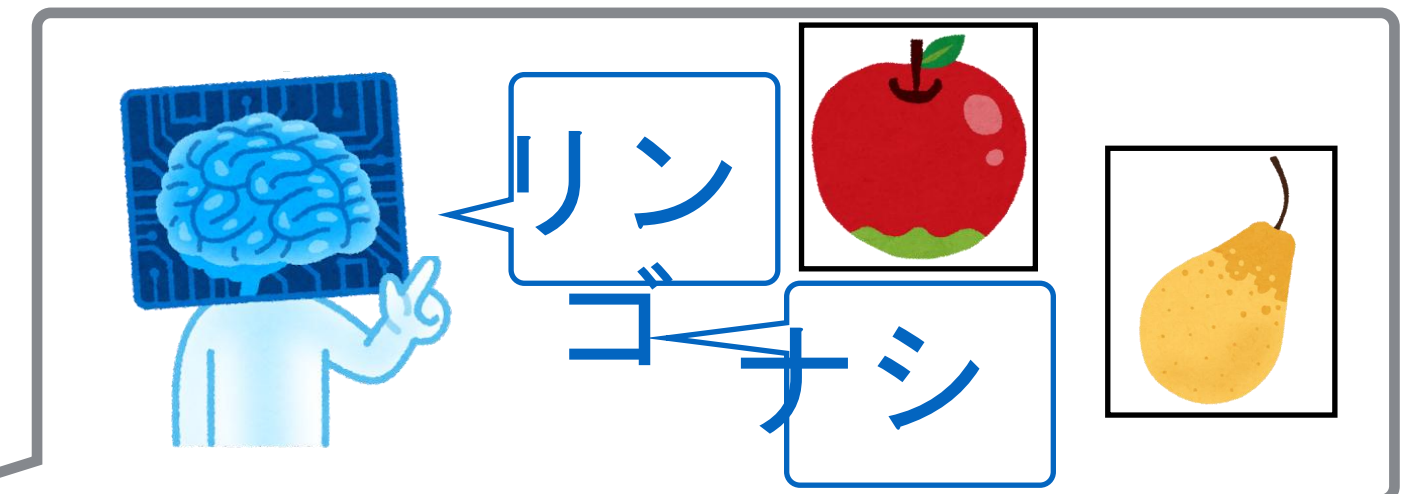
Random Forest

- 決定木（弱仮説）を T 個使って多数決で予測
 - ◆ 各決定木は説明変数（特徴量）を入力しクラスを予測
- ただし各決定木は N 個の特徴量のうちランダムに選んだ k 個の特徴量を用いる
 - ◆ 経験的に, $k = \sqrt{N}$ とされることが多い

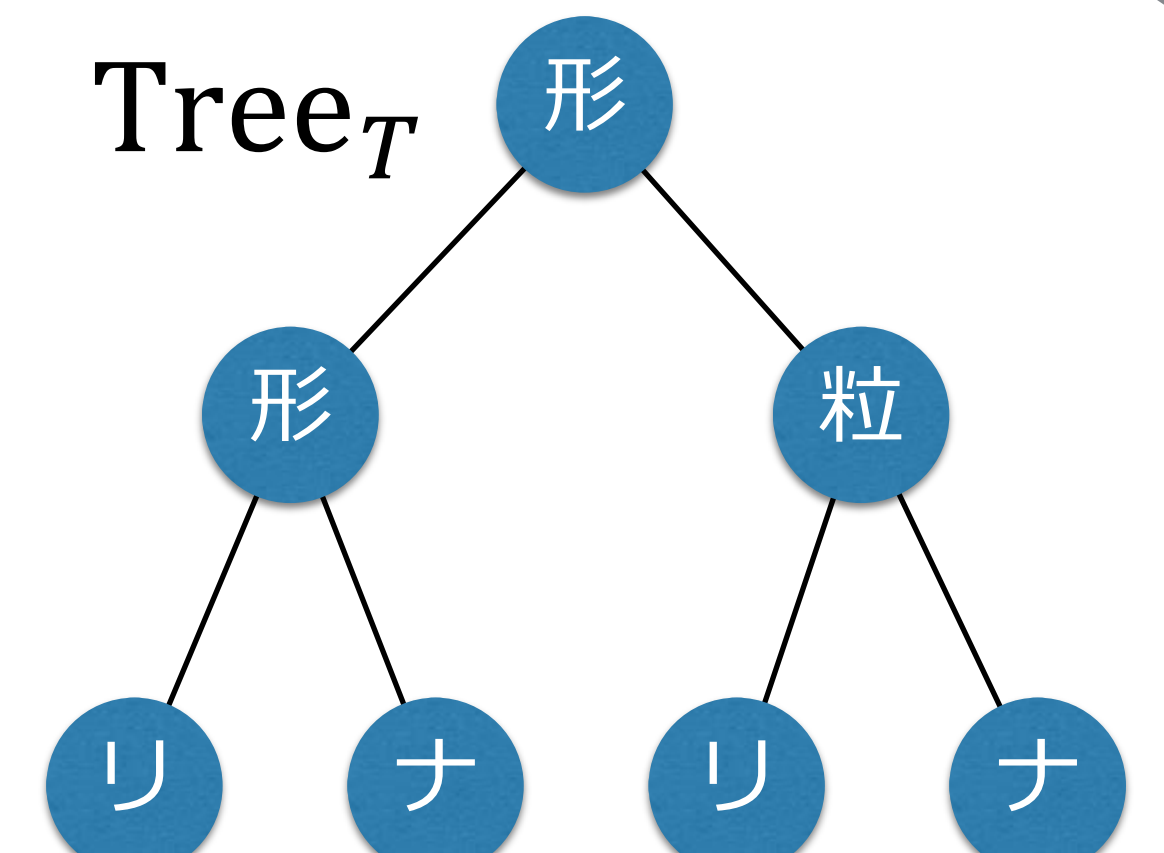
色と形だけ

色と艶だけ

全部の多数決で予測

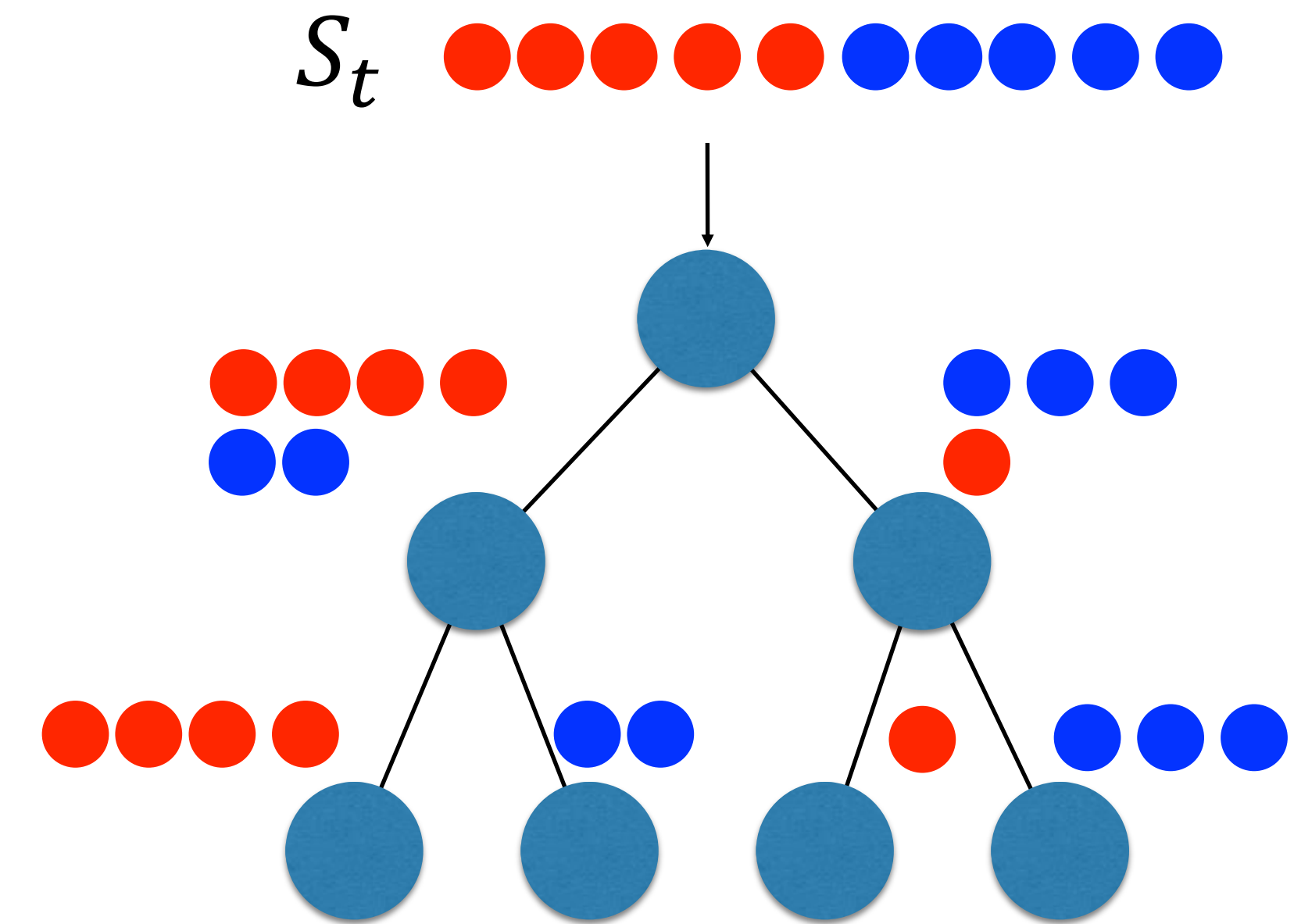


...



決定木はどうやって学習される？

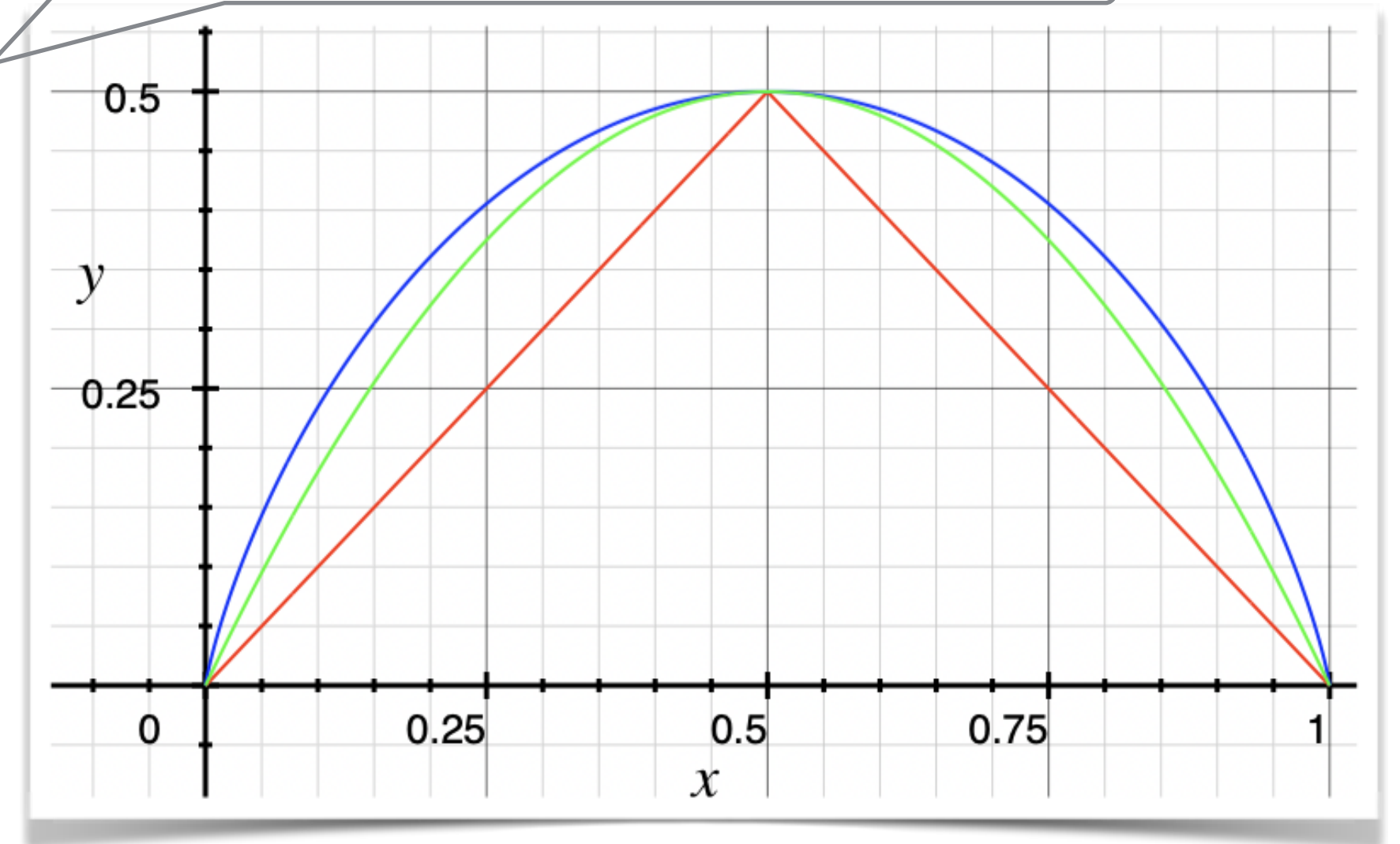
- 入力： S_t （全学習データの中から p 個を部分的にとってきたもの）
- アルゴリズム：
 - ◆ 最も「純度（purity）」が良くなるようなある説明変数を使った分割を探す（例： $x_3 \leq 3$ ）
 - ◆ 分割してさらに部分サンプル集合を作成
 - ◆ 各部分サンプルに対して上記を繰り返す
 - ◆ 「純粹」になったら終了



Impurity (不純度)

- 様々な尺度あり
- 2クラス分類の例： x を正例の割合とすると
 - ◆ 誤分類： $\min(x, 1 - x)$
 - ◆ エントロピー： $-x \log_2(x) - (1 - x) \log_2(1 - x)$
 - ◆ Gini index： $2x(1 - x)$
- 回帰だと2乗誤差などが使われる

これを指標にする決定木を
C4.5などと呼ぶ



これを指標にする決定木を
CARTなどと呼ぶ

基本的にはC4.5とCARTが主流

Random Forest のメリット・デメリット

■ メリット

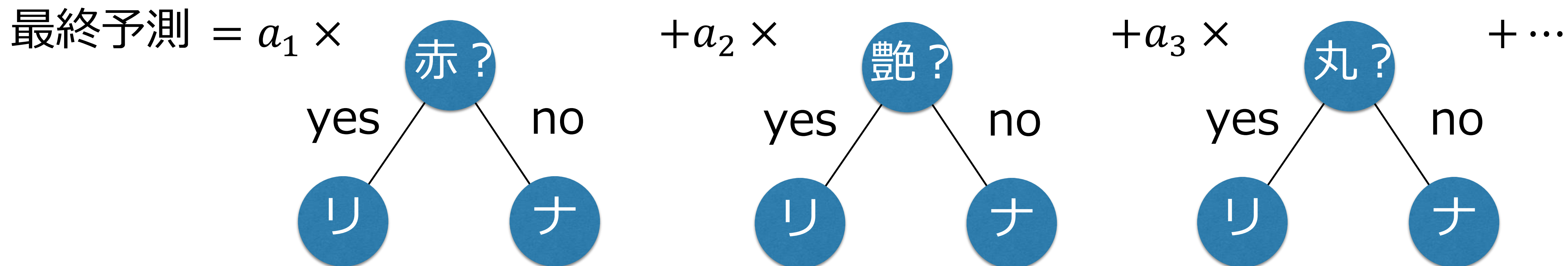
- ◆ 並列化可能なため速い
- ◆ 正規化などが不要
- ◆ カテゴリカル・データ（離散値）も扱える
- ◆ 欠損値も扱いやすい
 - 欠損値がないやつだけで分割したりする動作が容易
 - 欠損値に対してはエントロピーの計算を行わないなど

■ デメリット

- ◆ 木が深いと過学習する可能性が高まる & 計算効率が落ちるので、ちょうどよい値に設定する必要あり（ハイパーパラメータ）
- ◆ 分類に関係ない特徴量が多いと性能が落ちやすい（例えばある木に割り当てられた特徴量が全部無関係な特徴量だとその木は...）
- ◆ 特徴は学習できないので複雑なデータの場合はあまり良い性能が出ない

Boosting

- Random Forest と同様, たくさんの予測モデル (弱仮説) をたくさん組み合わせて統合する学習器
 - ◆ 組み合わせる予測モデルには決定株 (深さ1の木) がよく使われる
 - ◆ 組み合わせ方の基本は「重み付き多数決」 ↓



学習の様子を見てみよう！

簡単のため、右のような10個のデータ

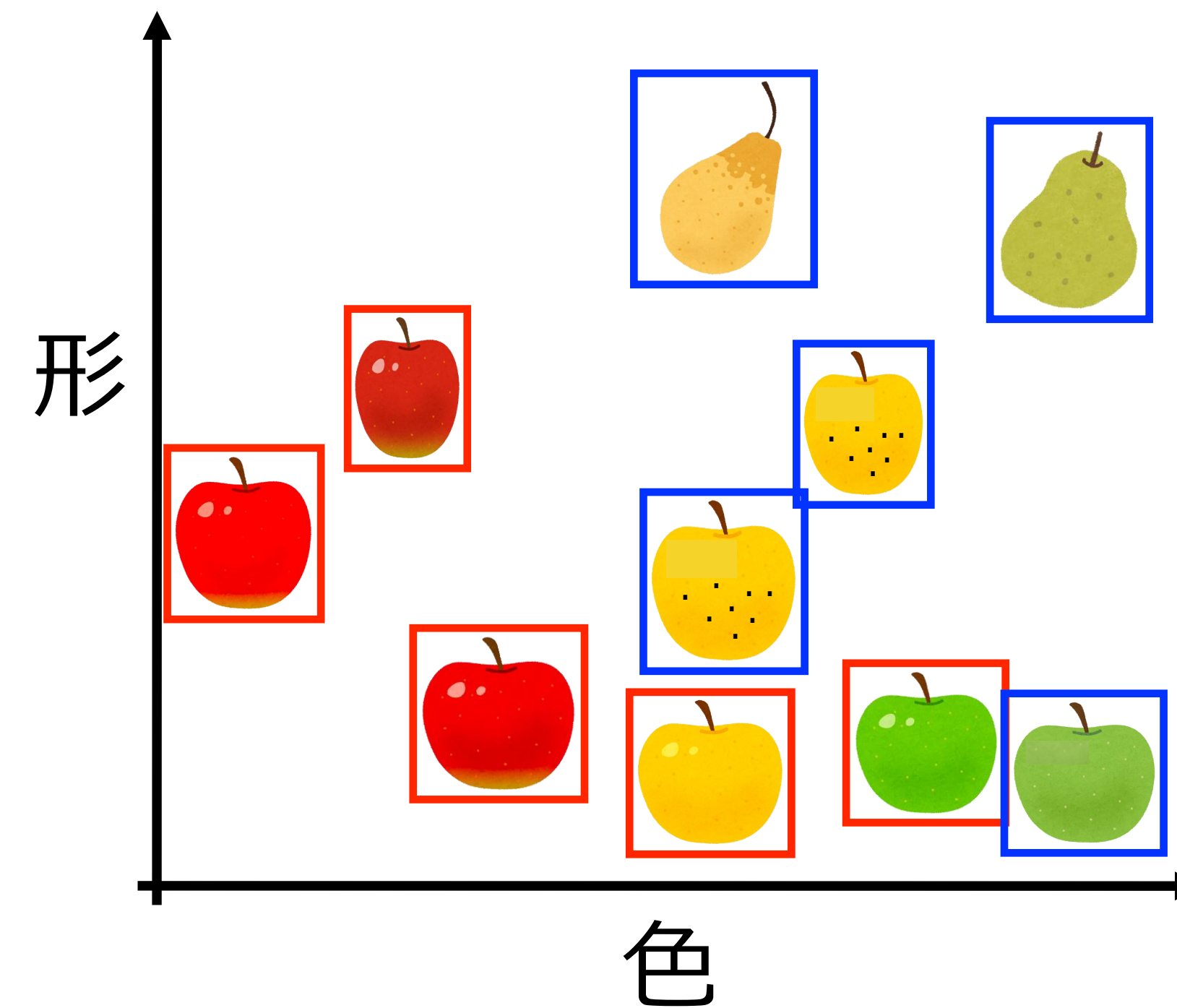
データ x は色の情報と形の情報のみ

弱仮説は以下の3つ：

「赤いのはリンゴ」

「丸くないのはナシ」

「緑褐色はナシ」



学習の様子を見てみよう！

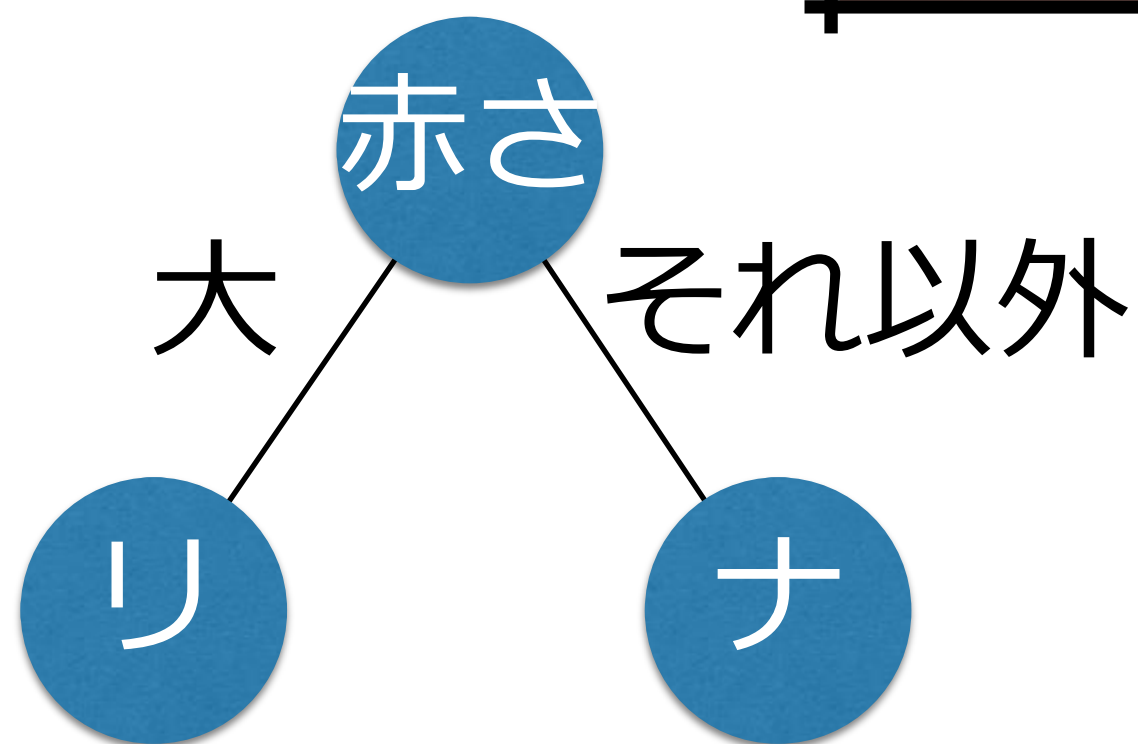
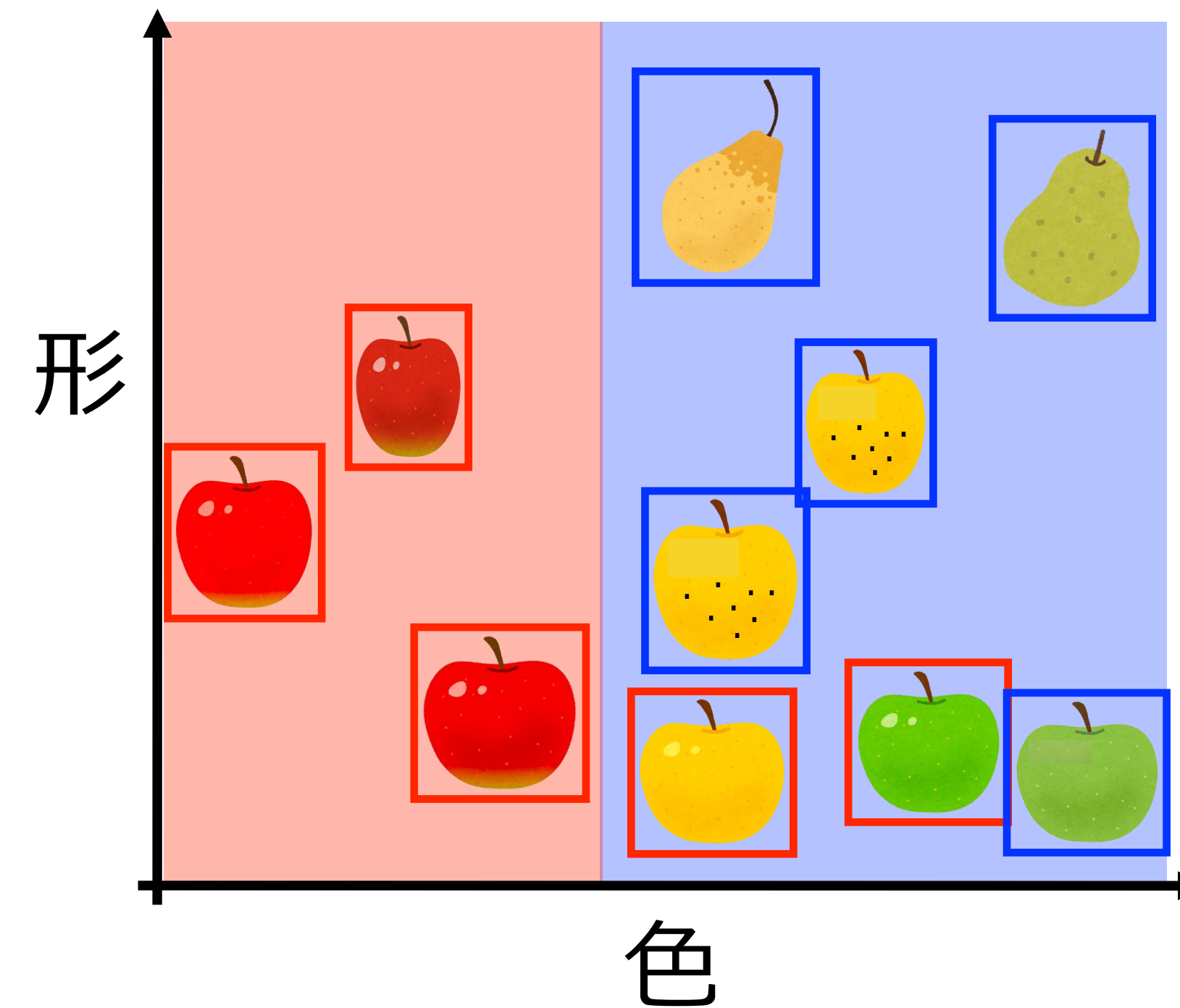
簡単のため、右のような10個のデータ

データ x は色の情報と形の情報のみ

弱仮説（特徴量）は以下の3つ：

「赤い \leftrightarrow 緑」

「丸さ」



学習の様子を見てみよう！

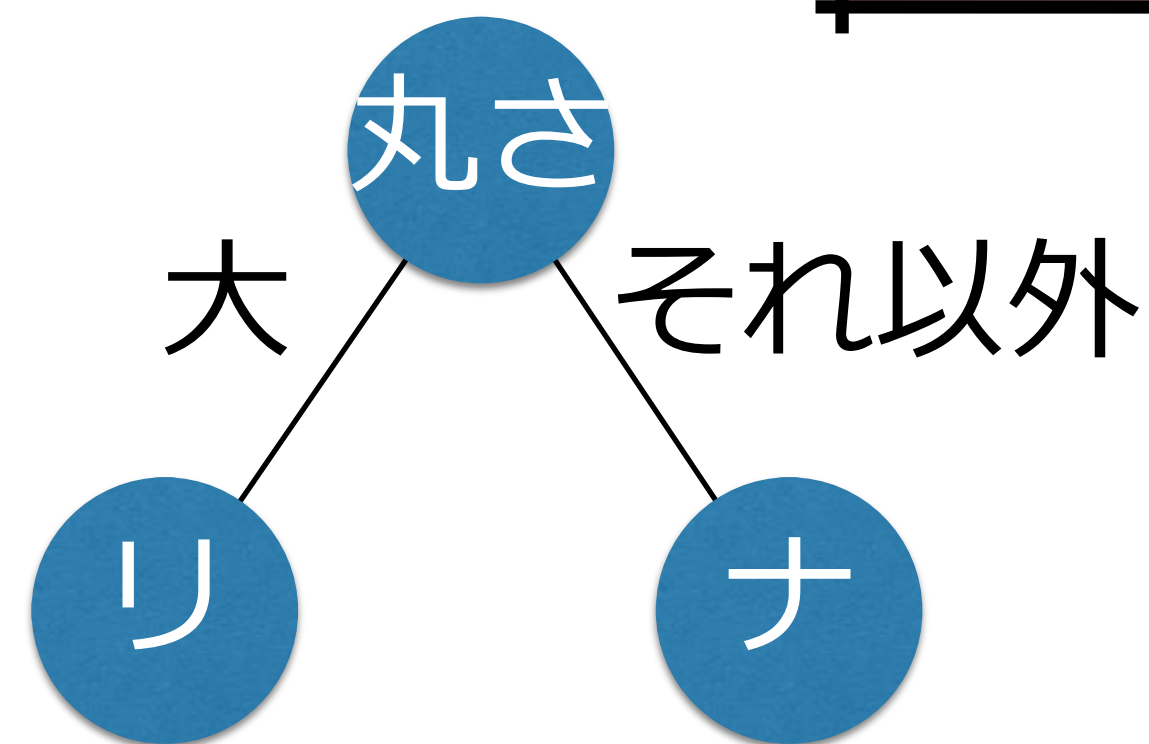
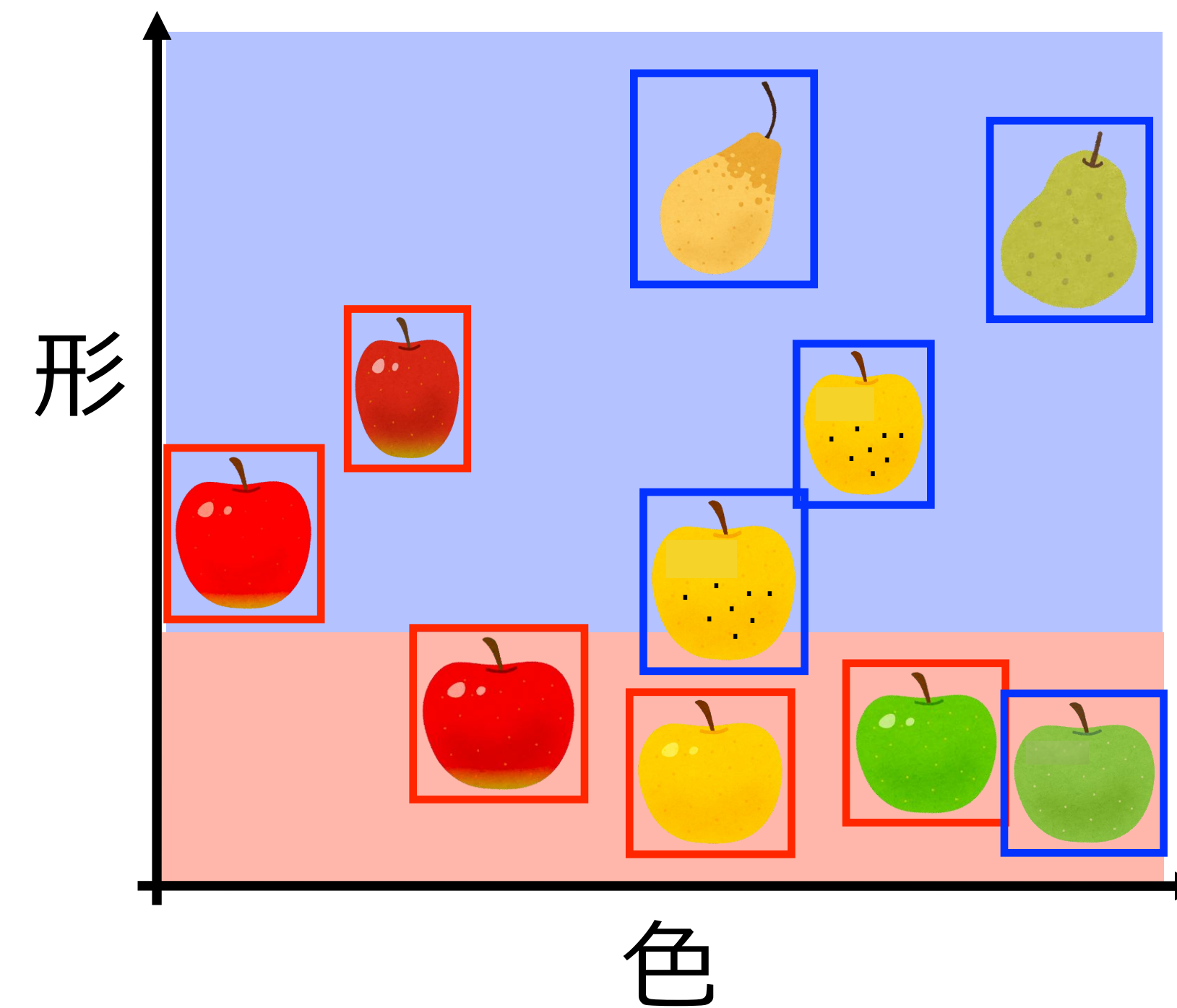
簡単のため、右のような10個のデータ

データ x は色の情報と形の情報のみ

弱仮説（特徴量）は以下の3つ：

「赤い \leftrightarrow 緑」

「丸さ」



学習の様子を見てみよう！

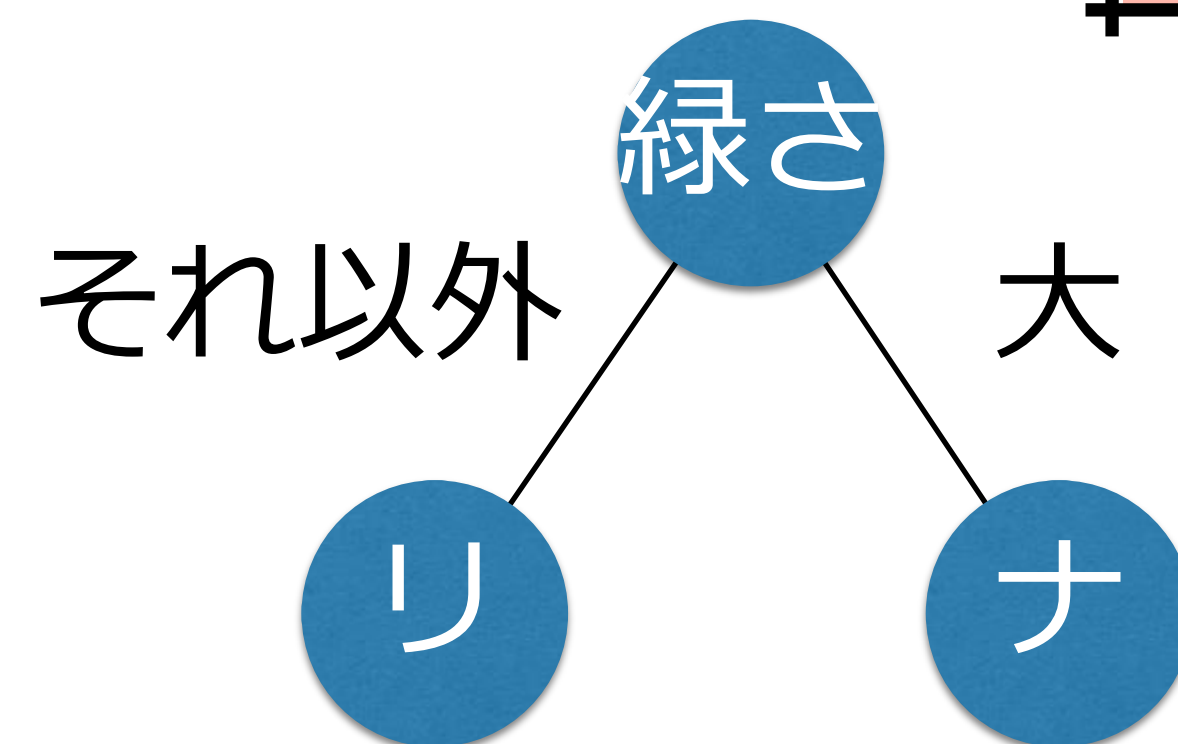
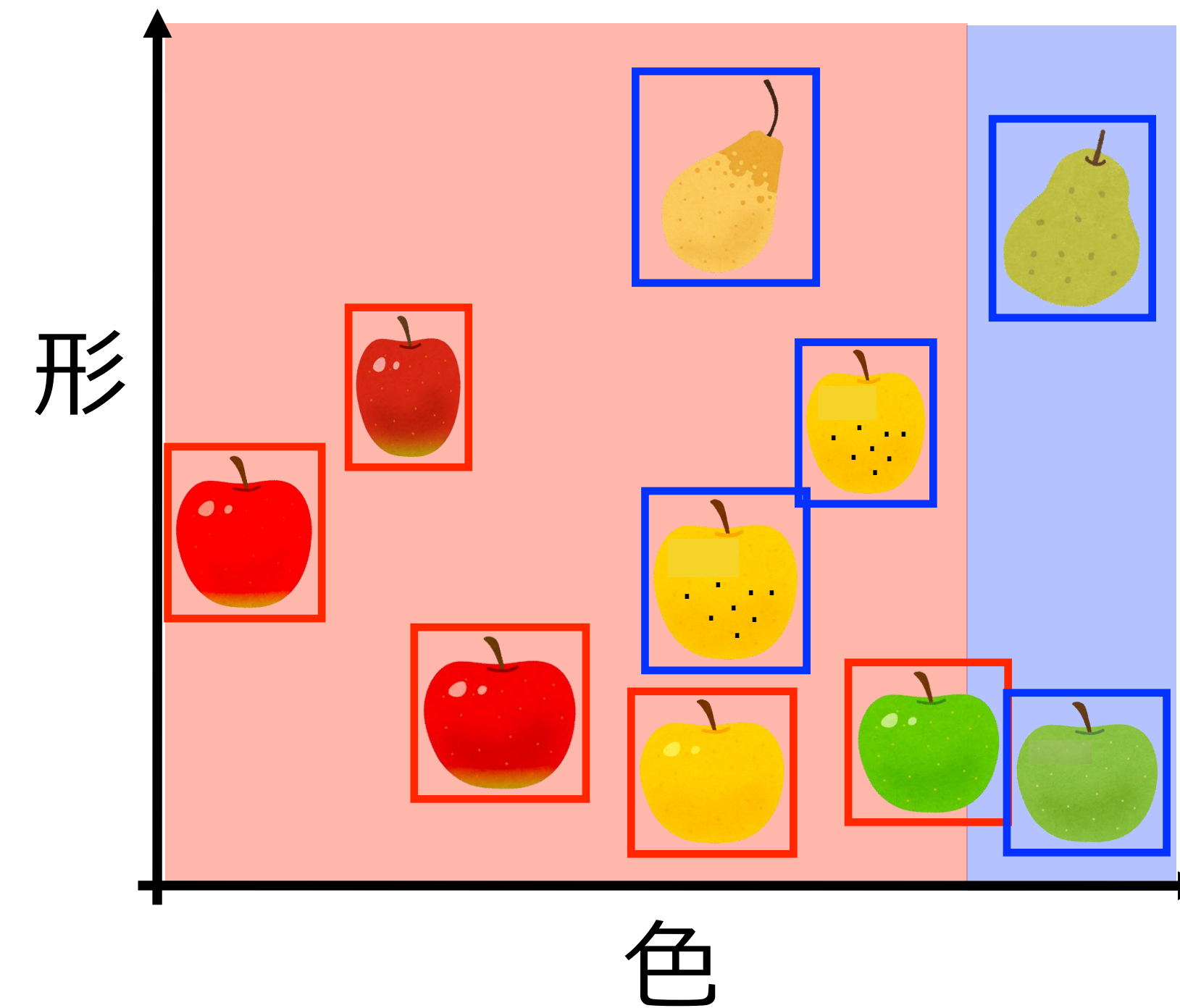
簡単のため、右のような10個のデータ

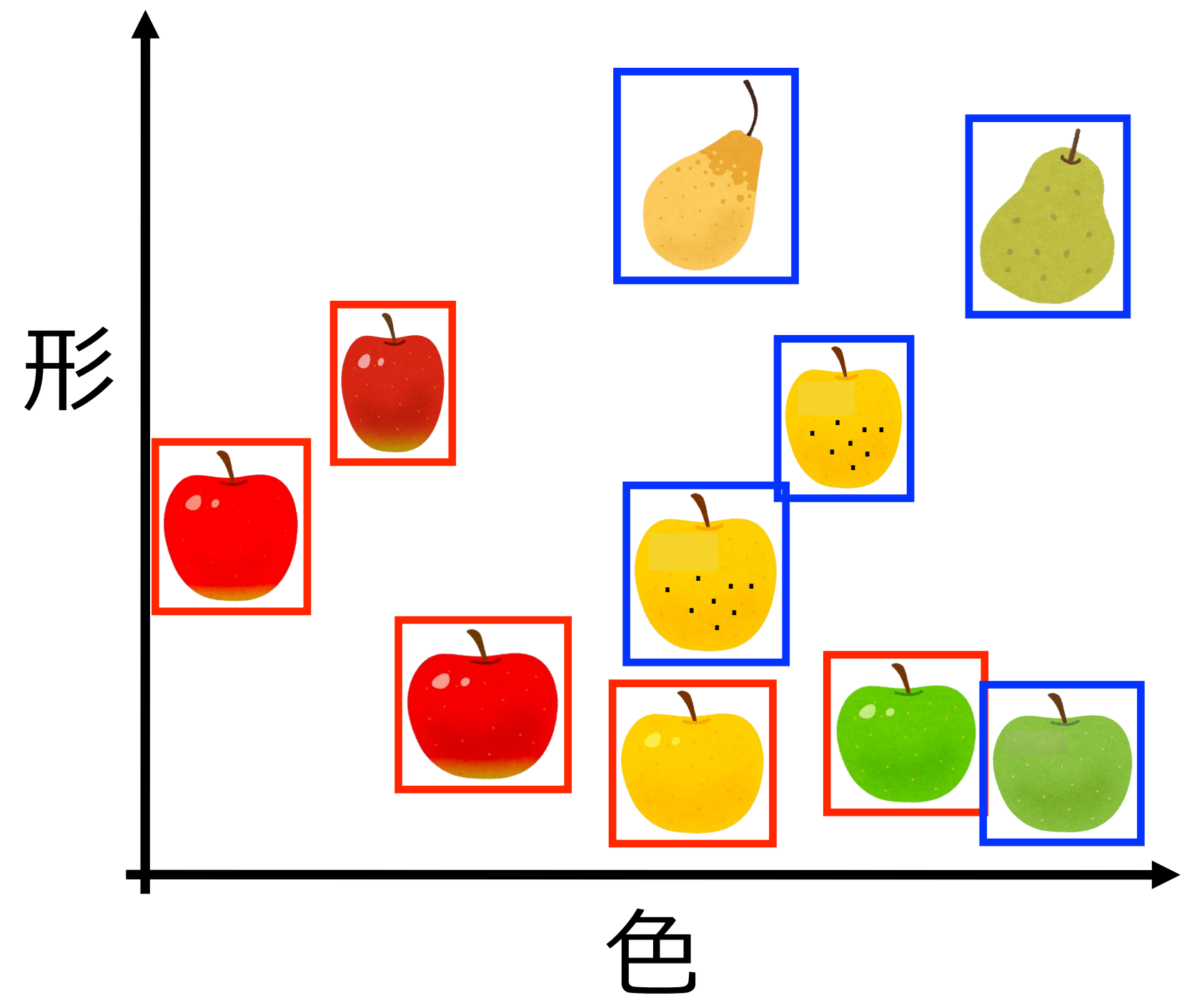
データ x は色の情報と形の情報のみ

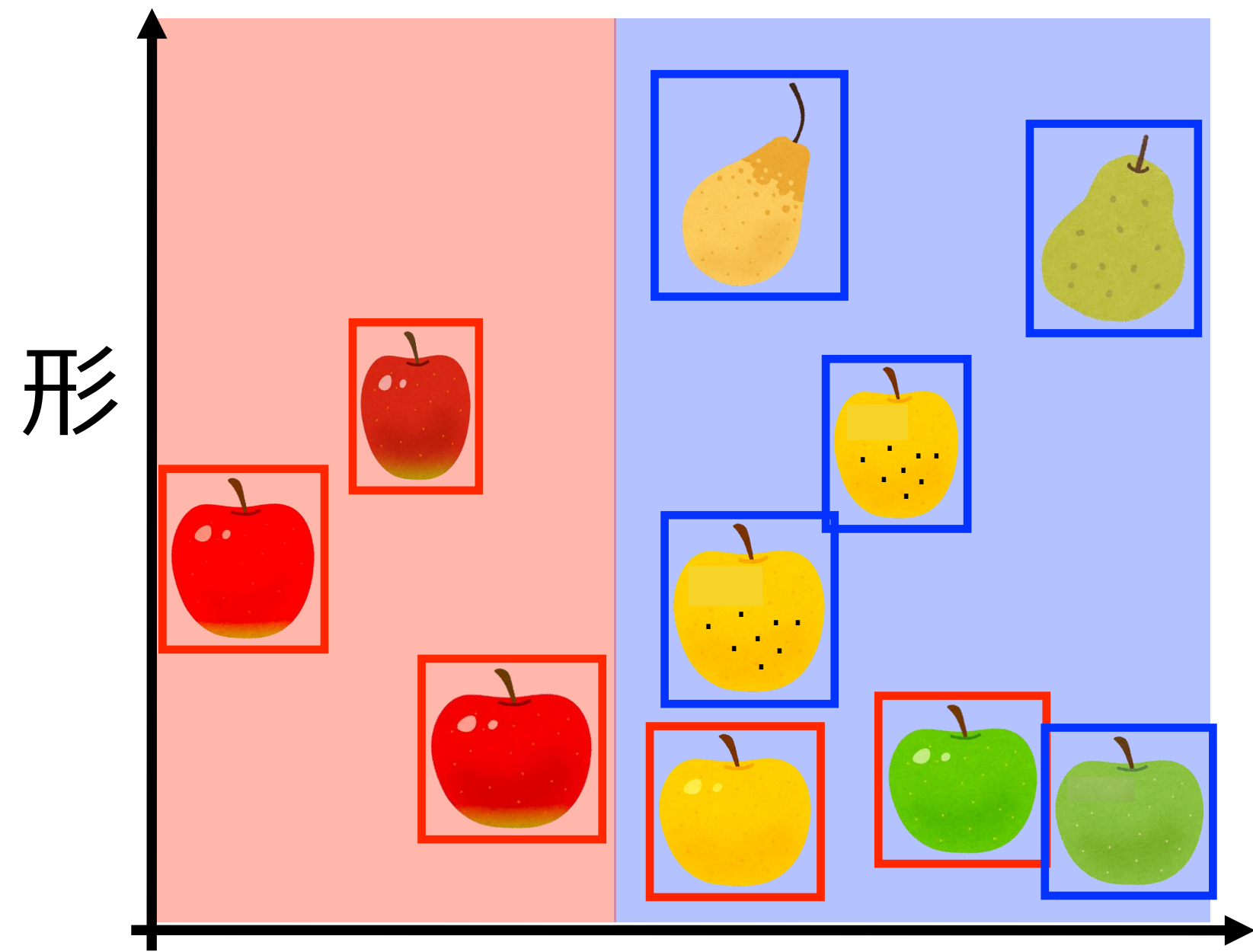
弱仮説（特徴量）は以下の3つ：

「赤い \leftrightarrow 緑」

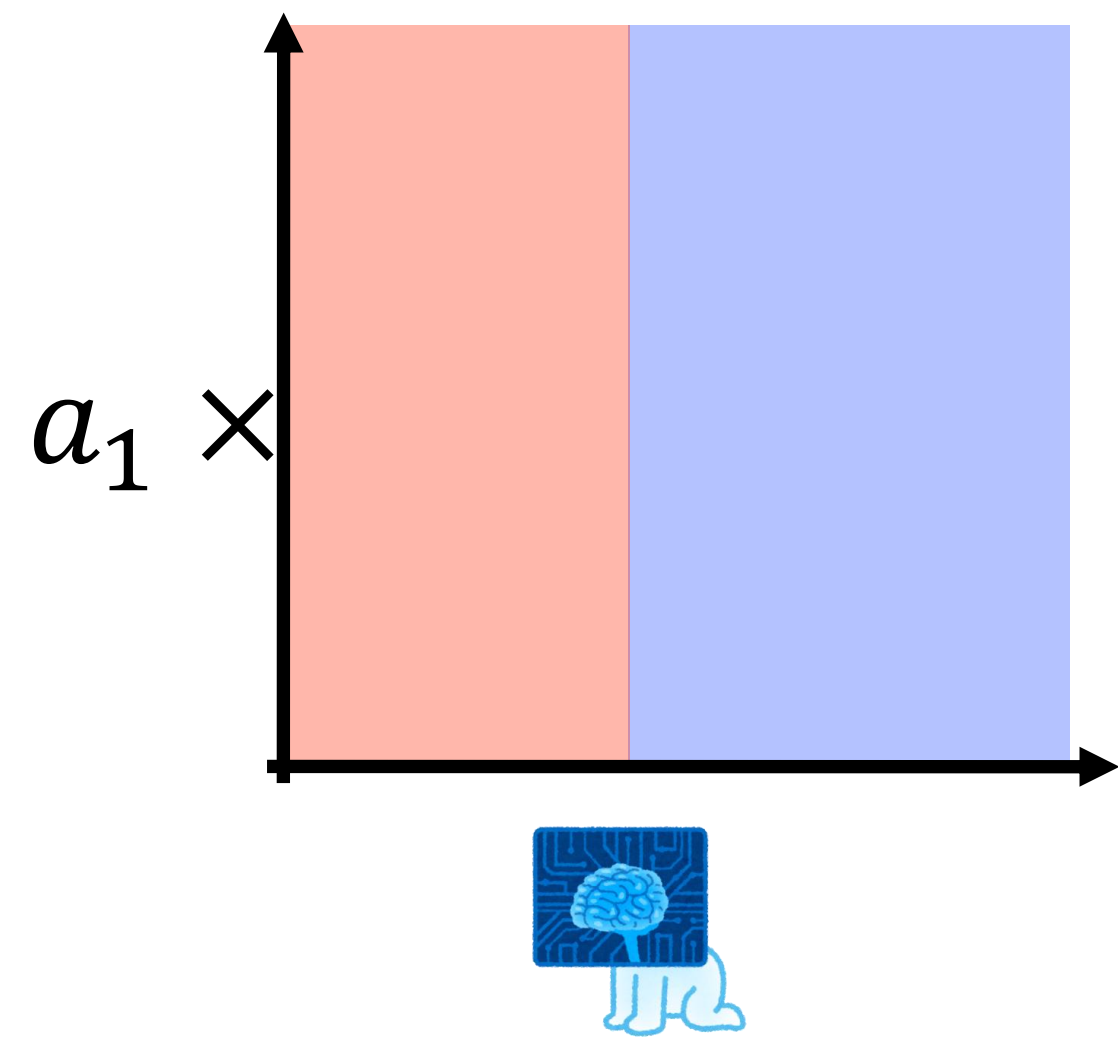
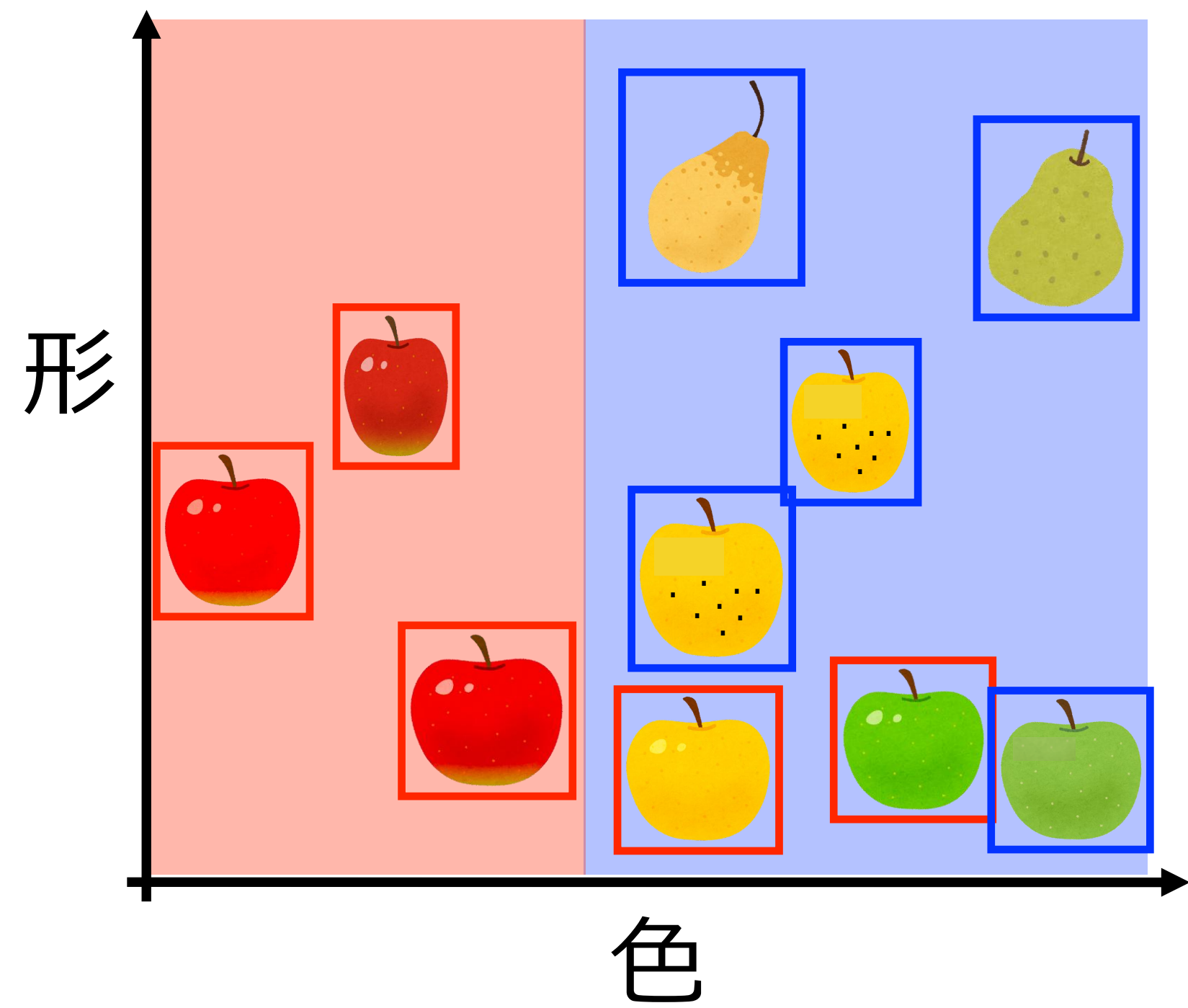
「丸さ」

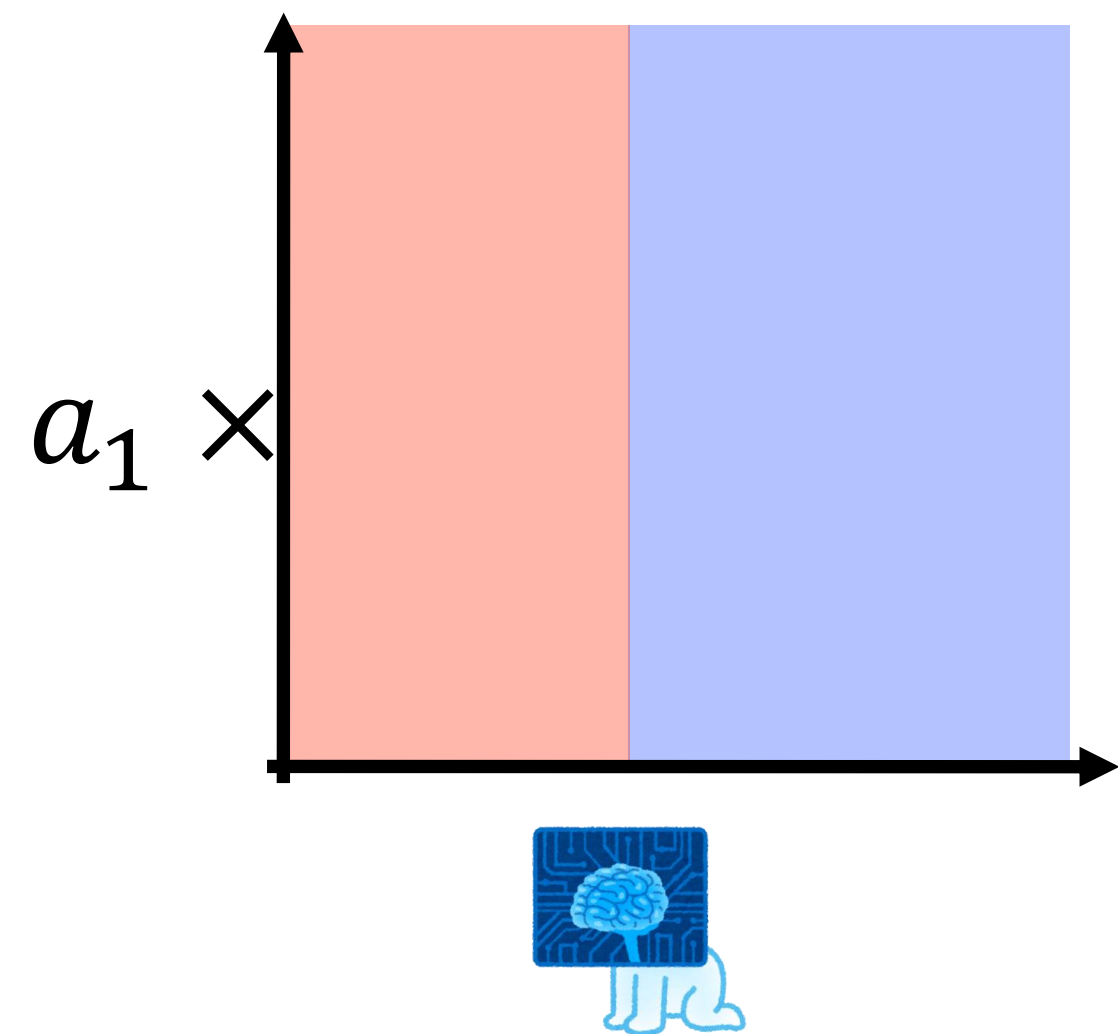
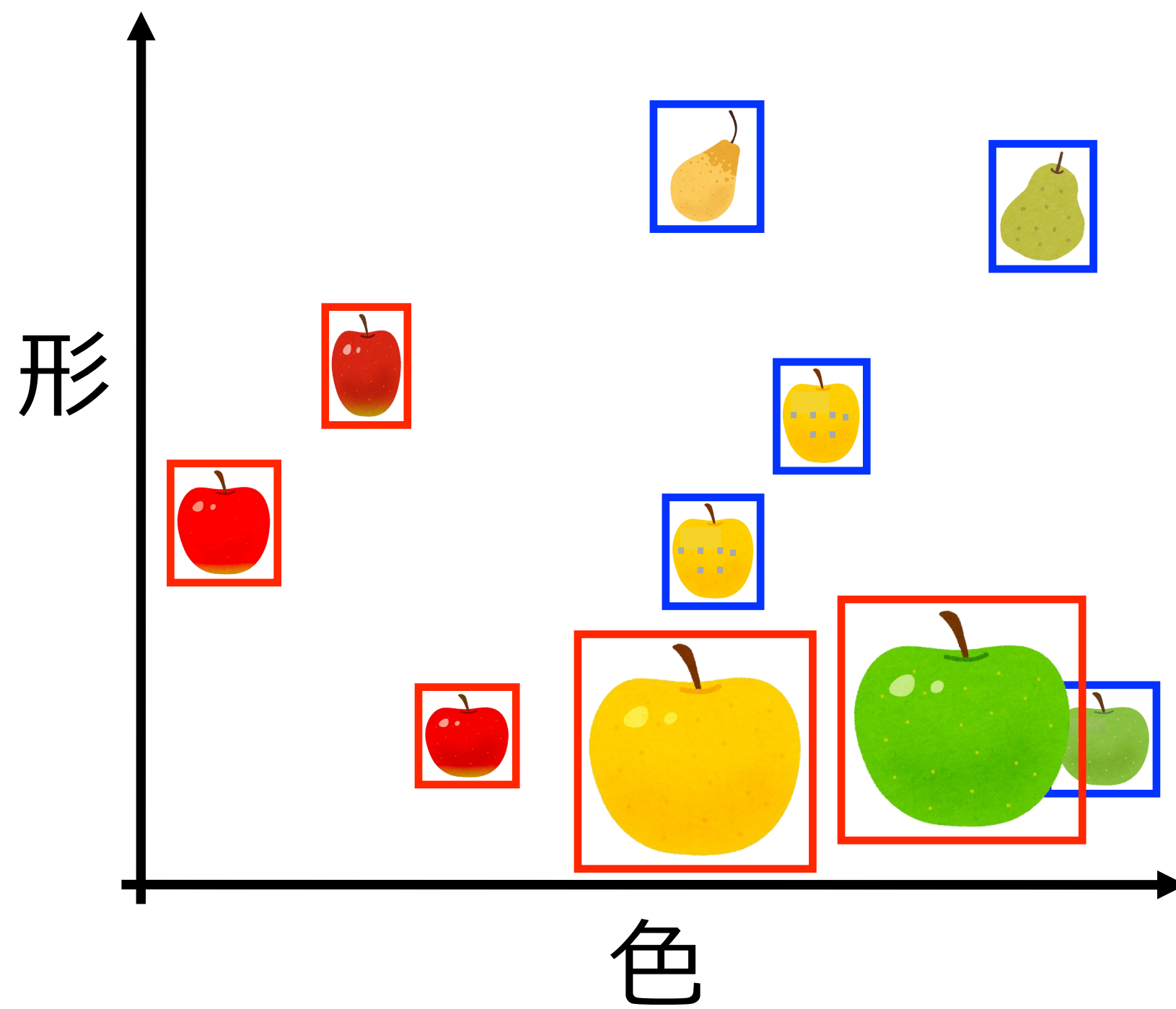
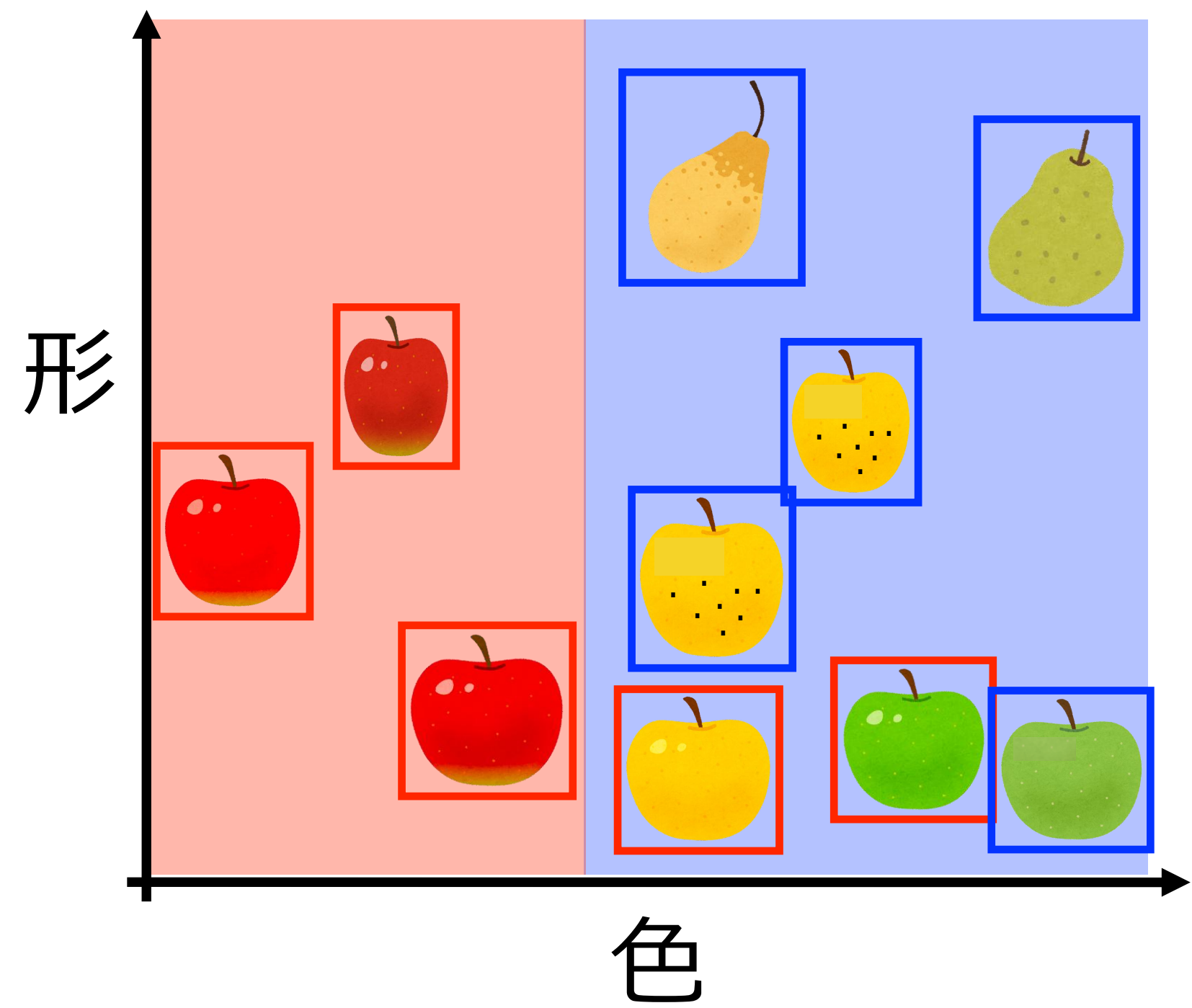


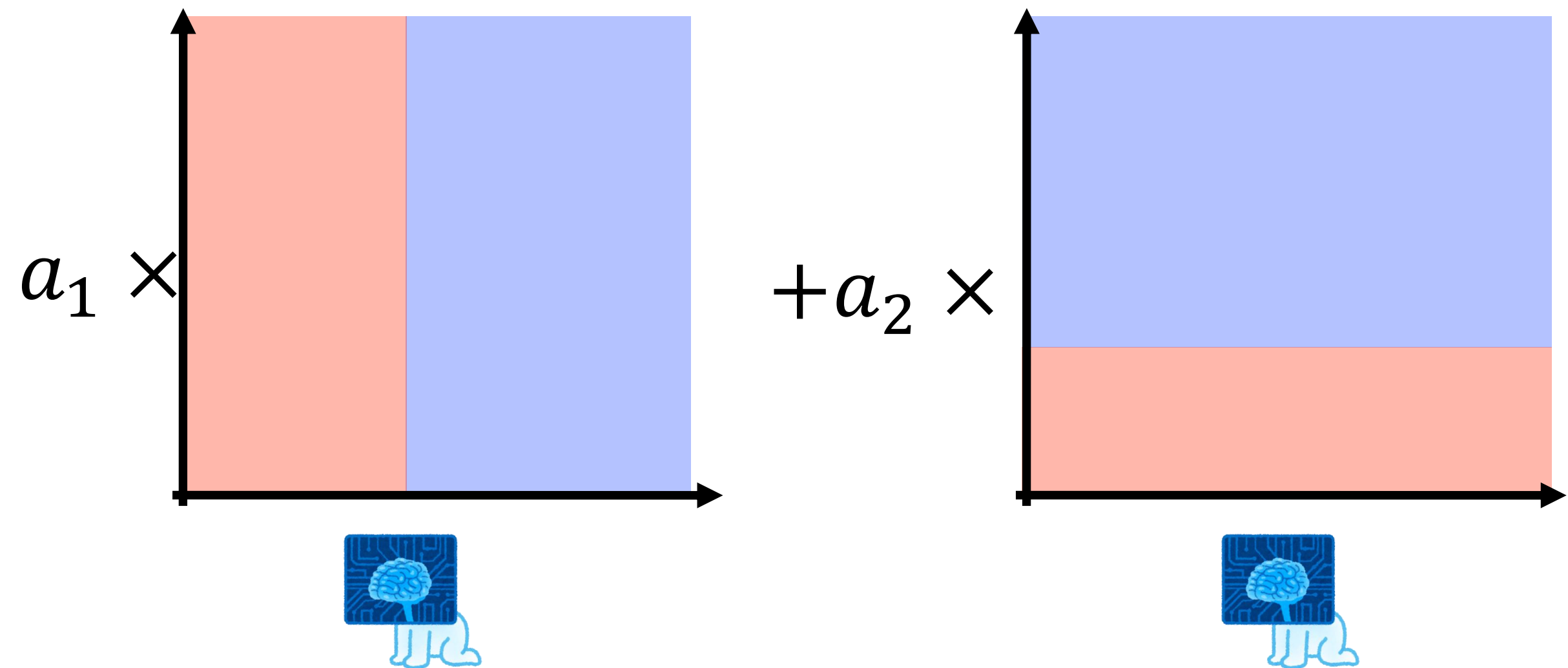
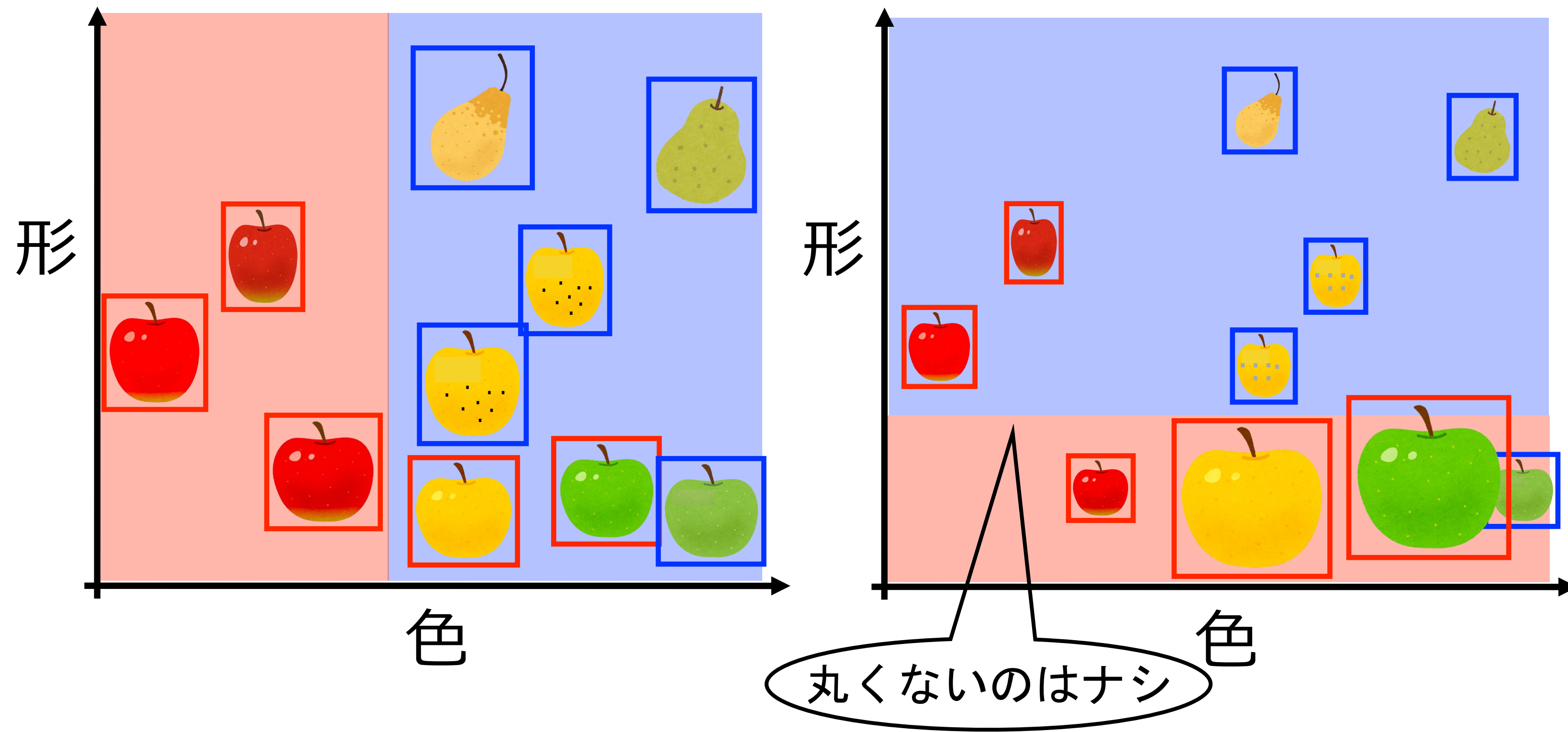


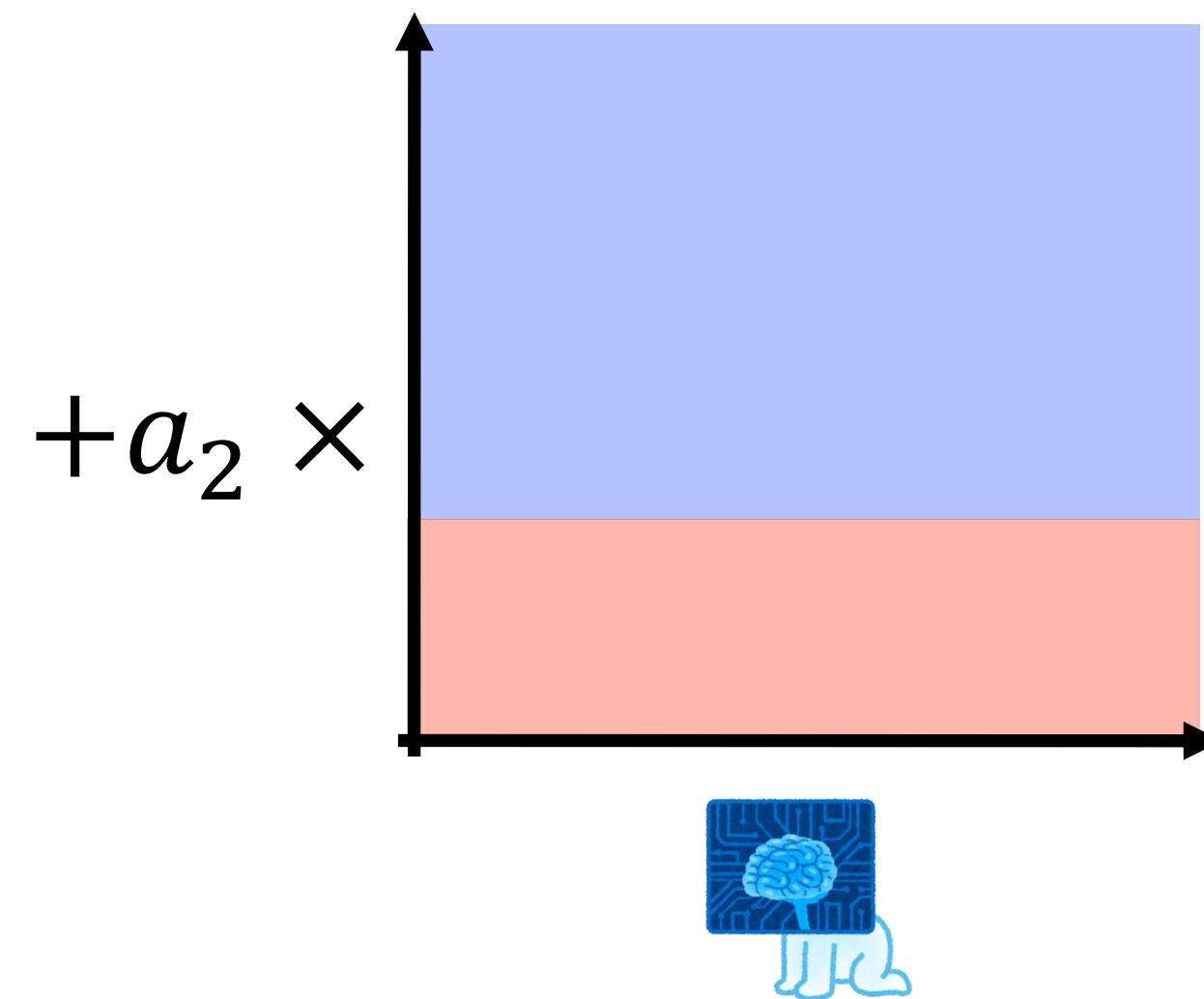
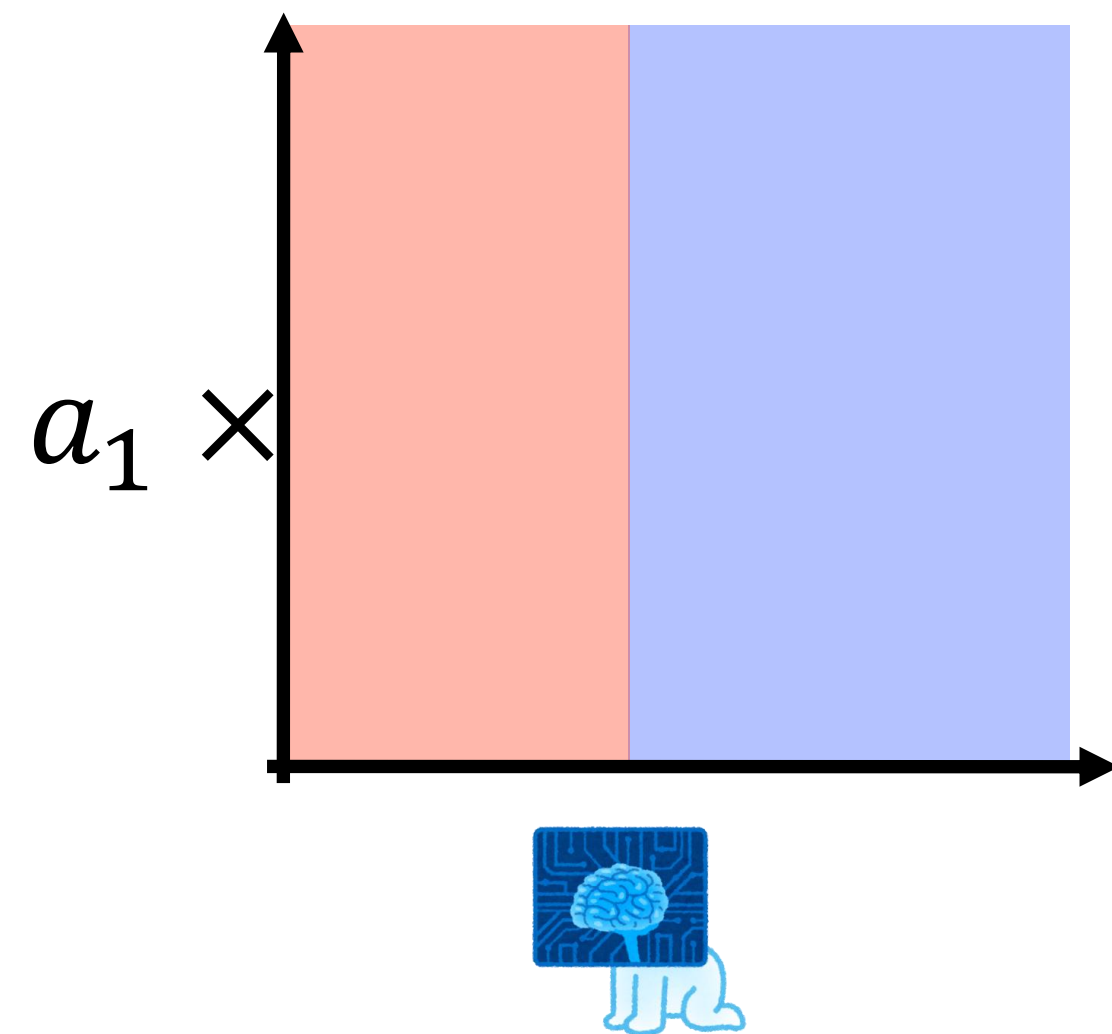
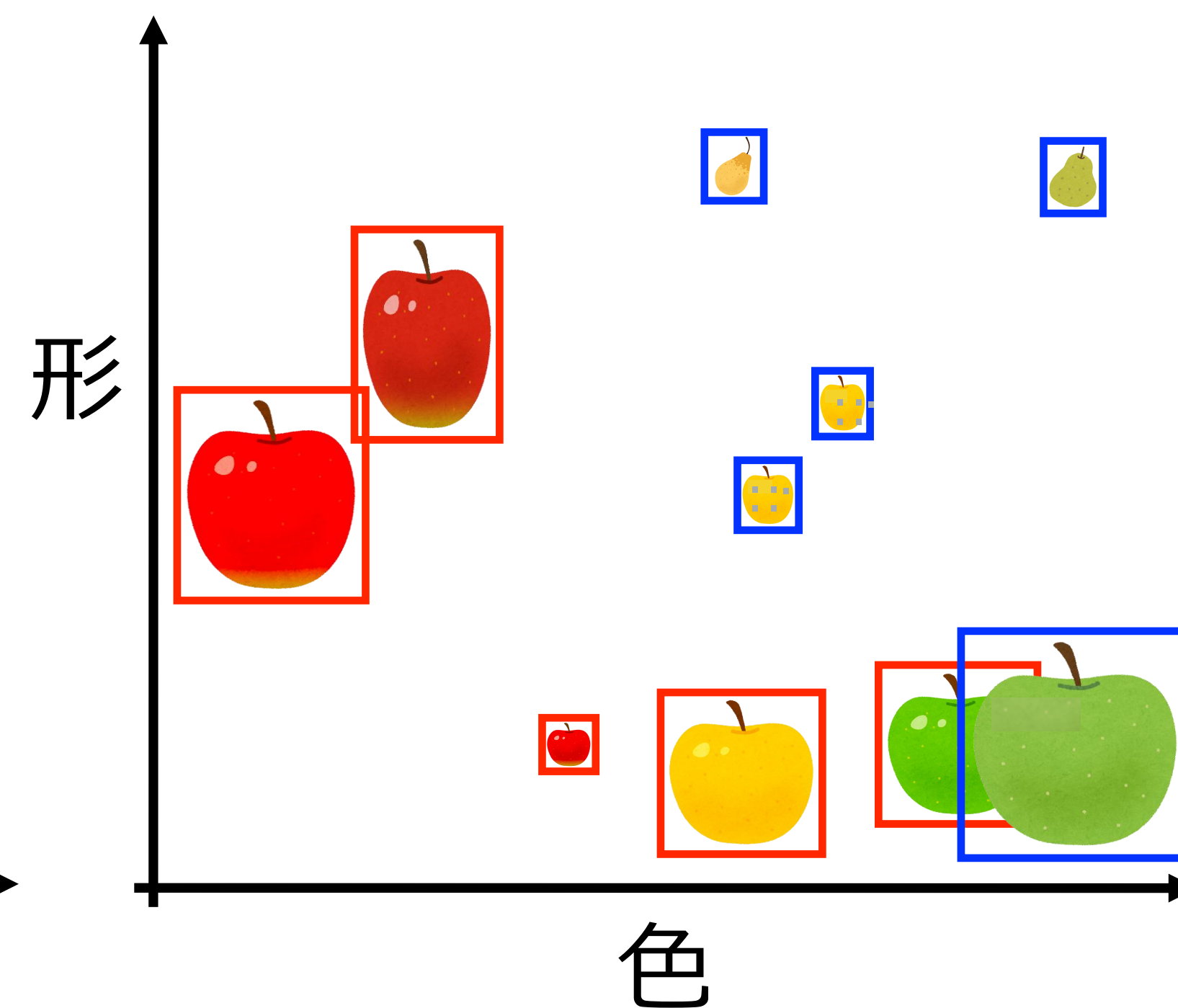
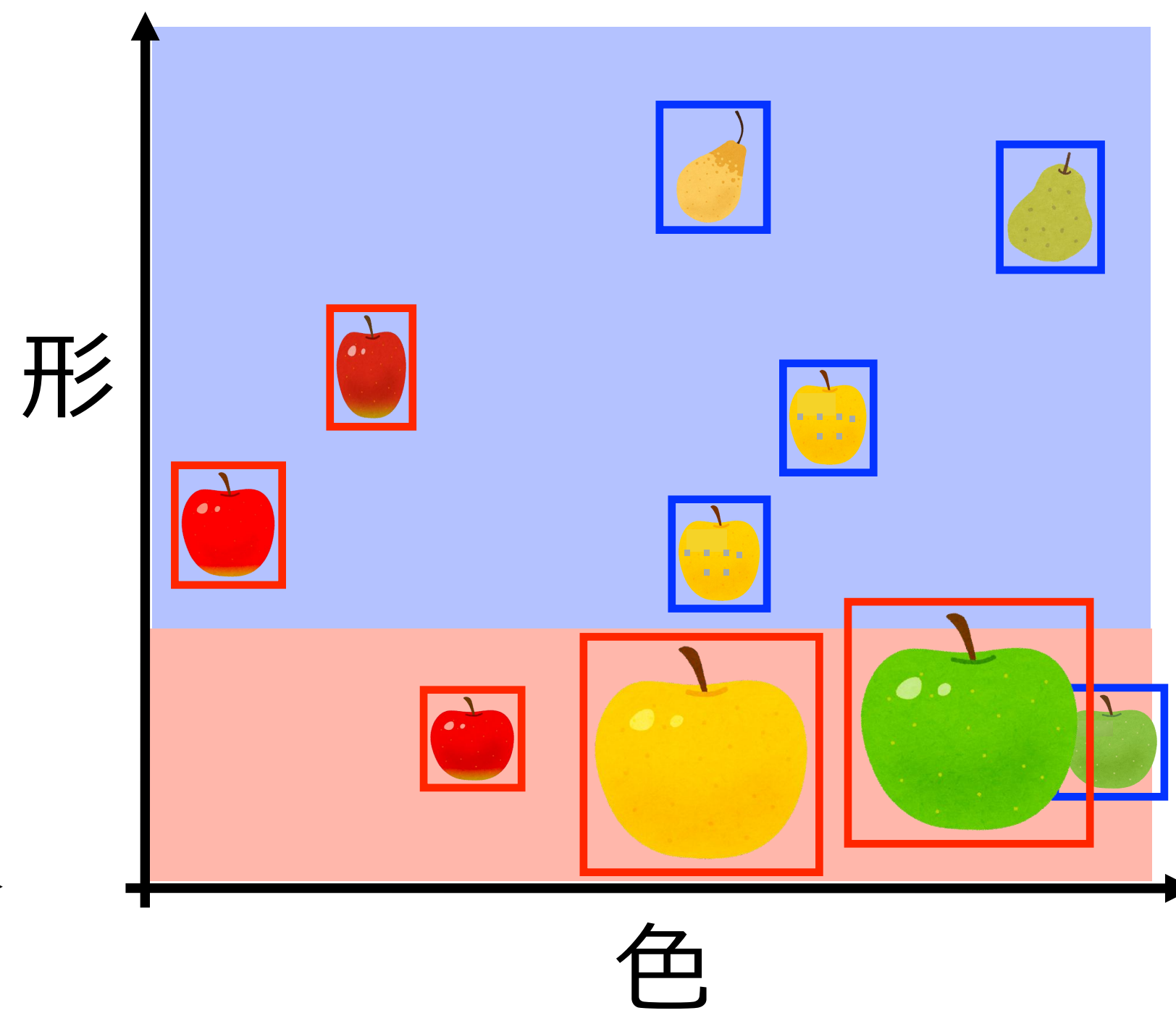
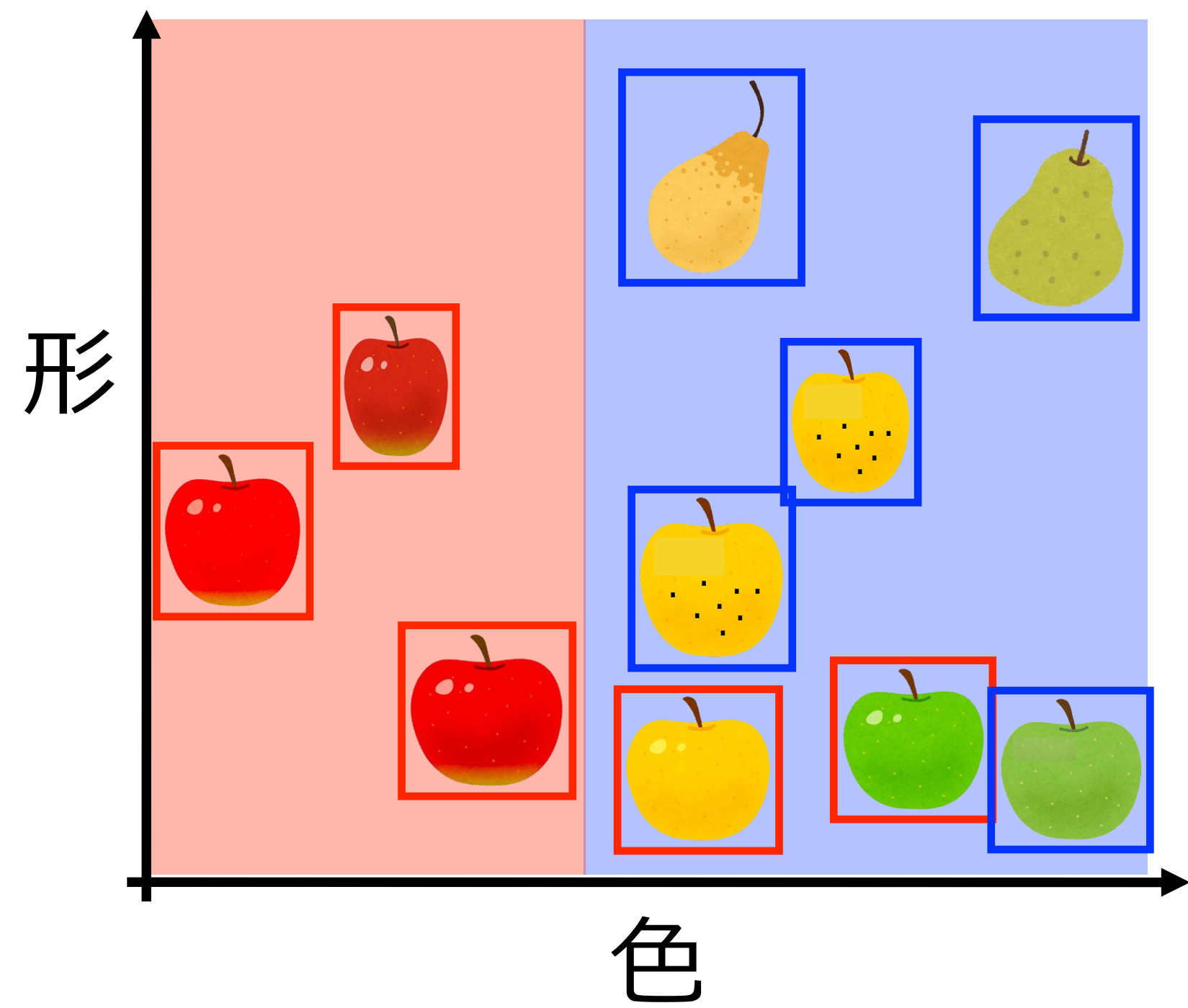


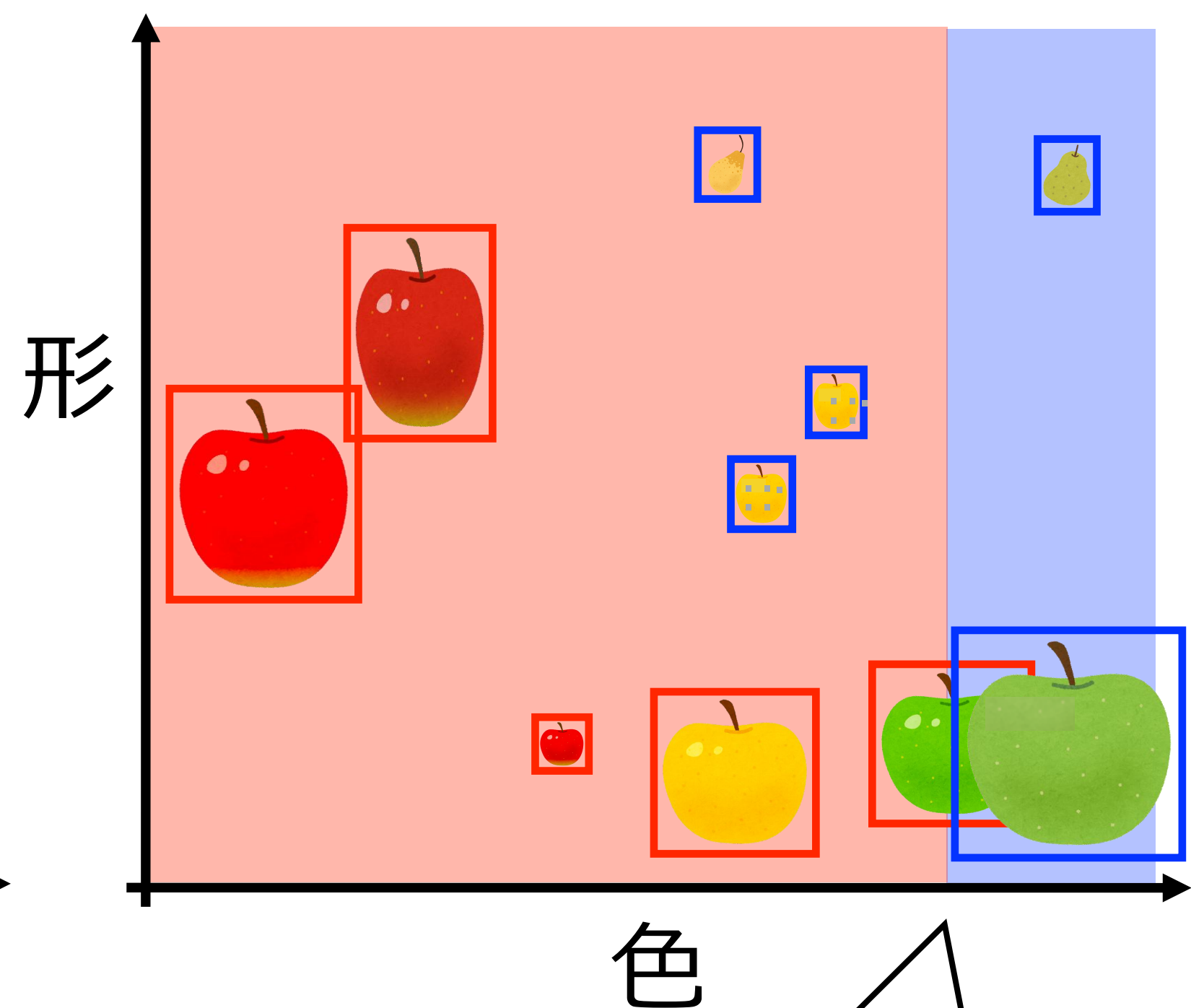
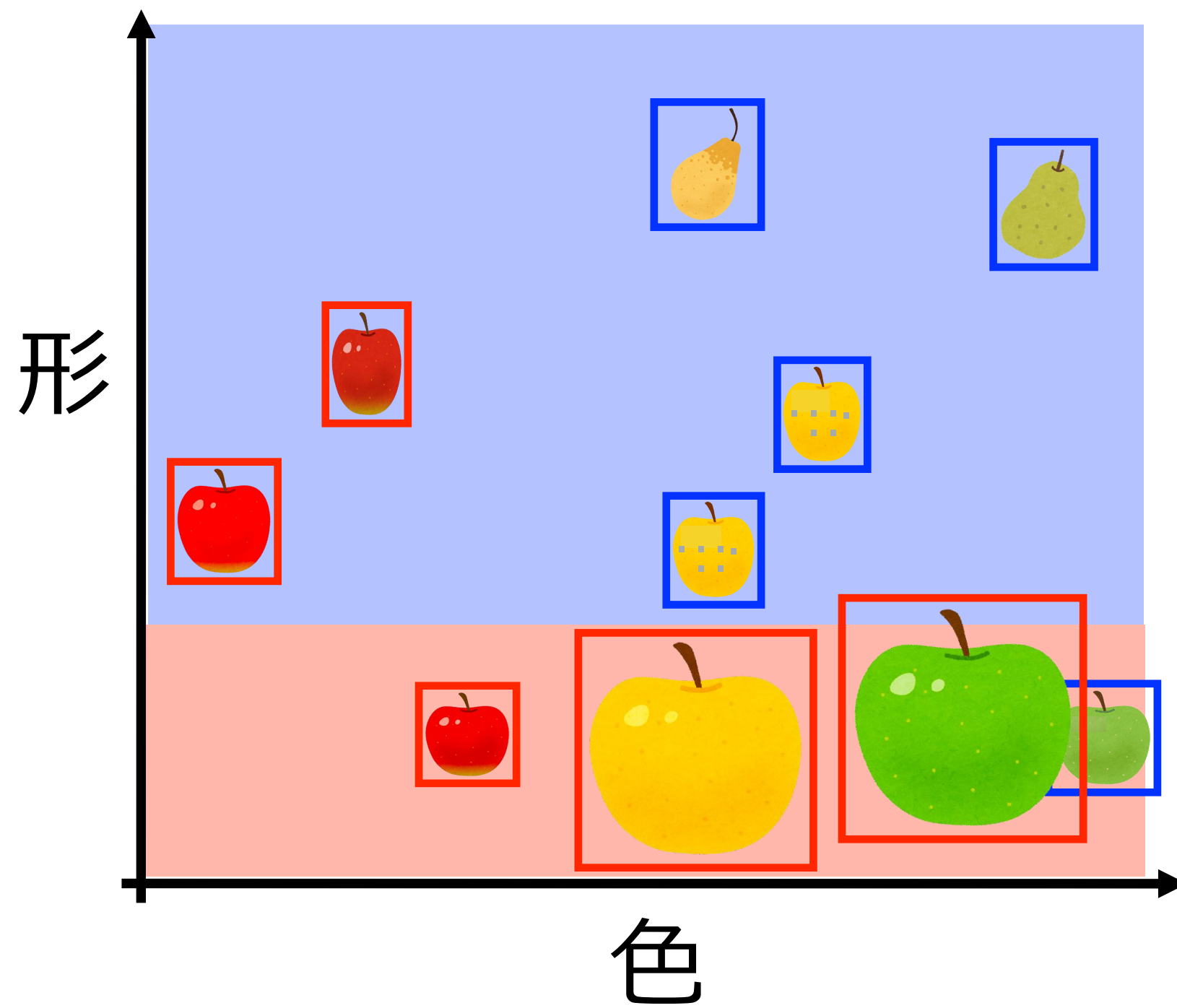
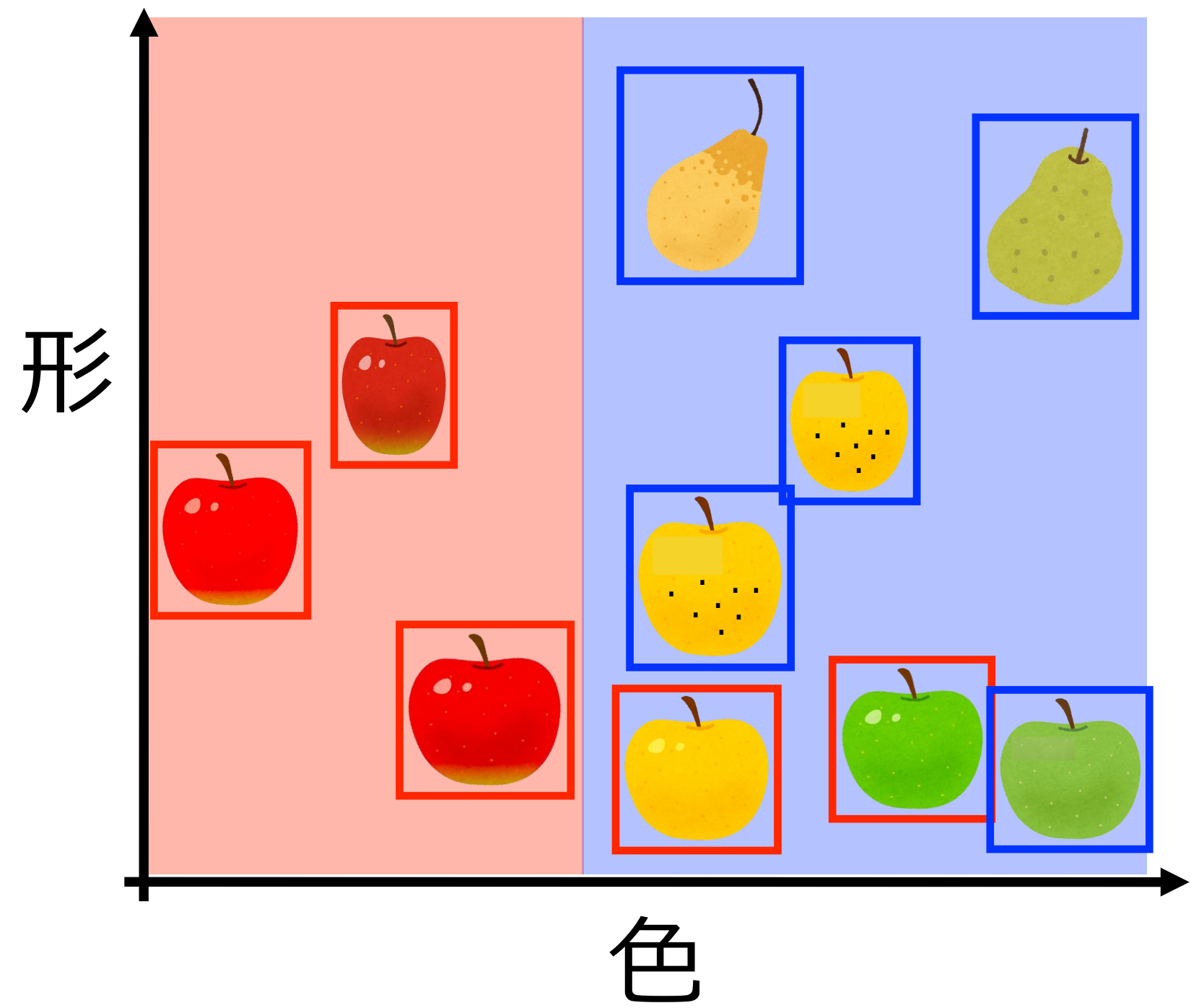
赤いのはリンゴ



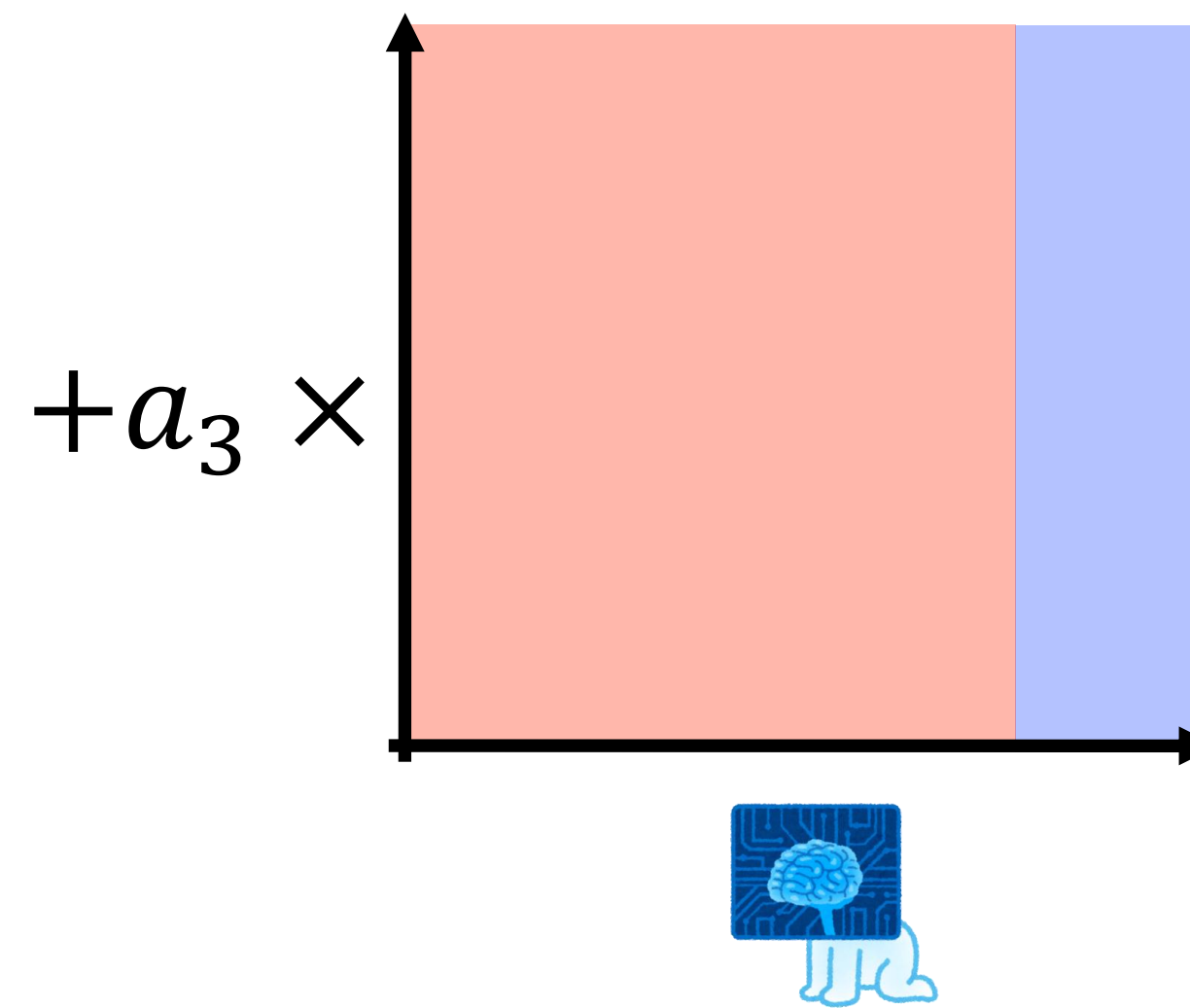
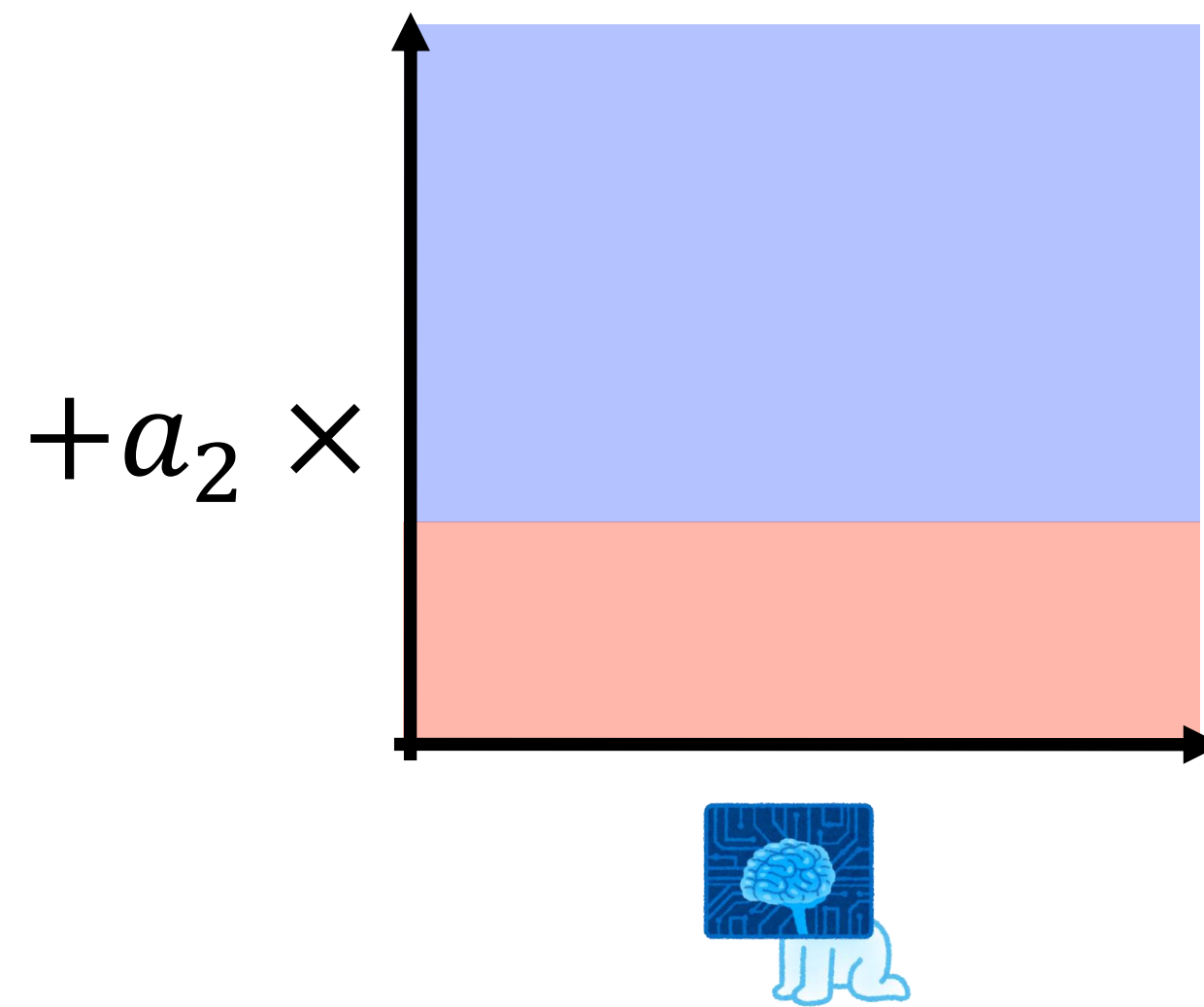
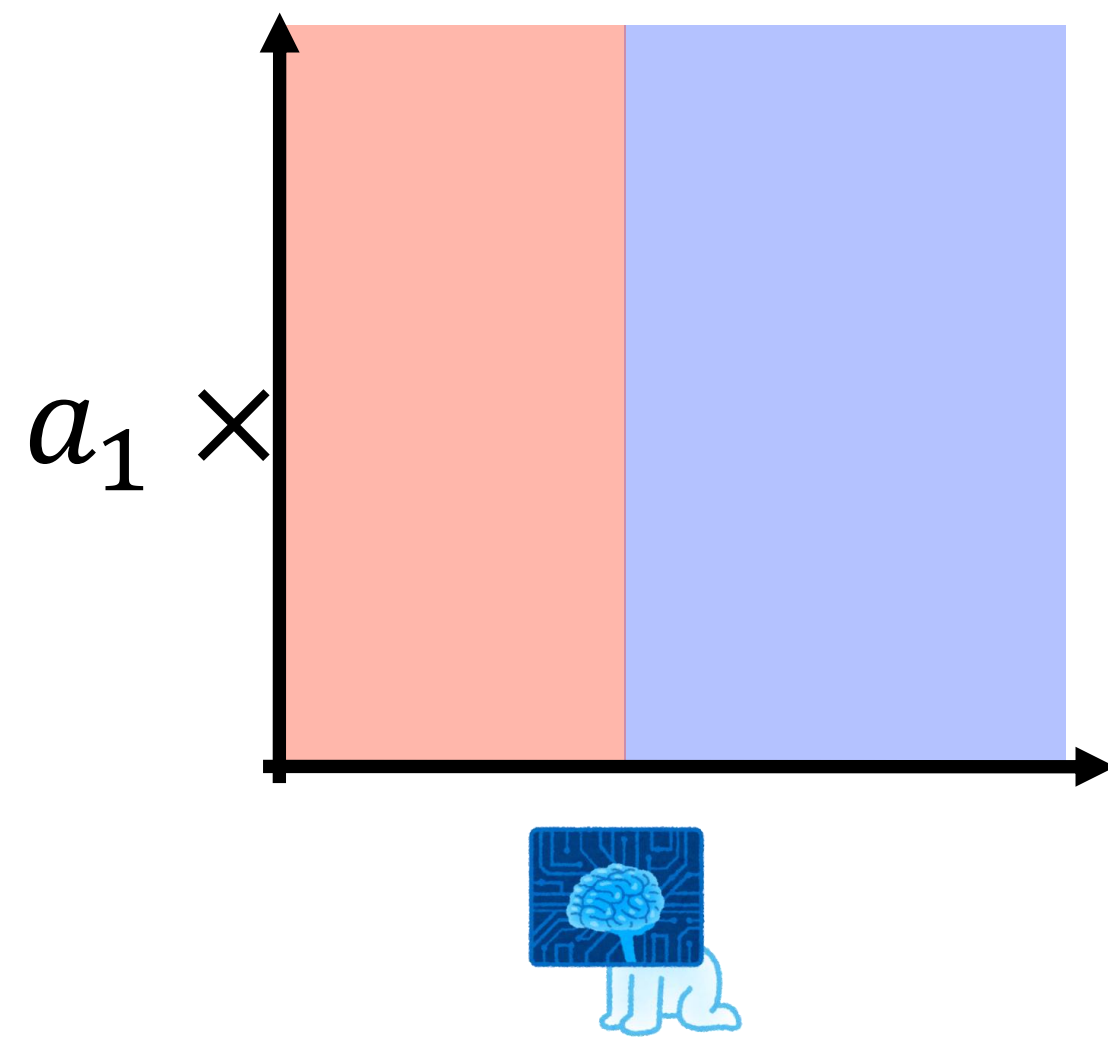


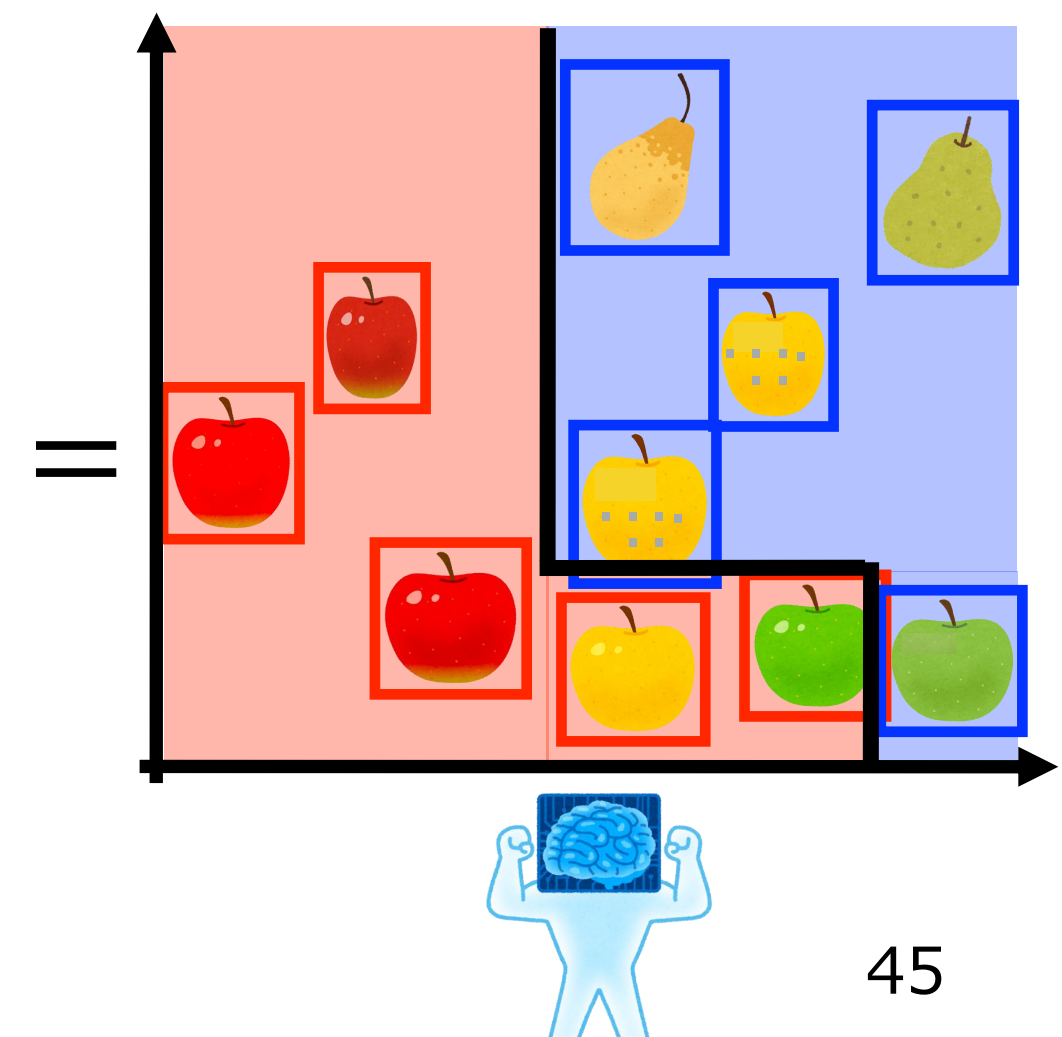
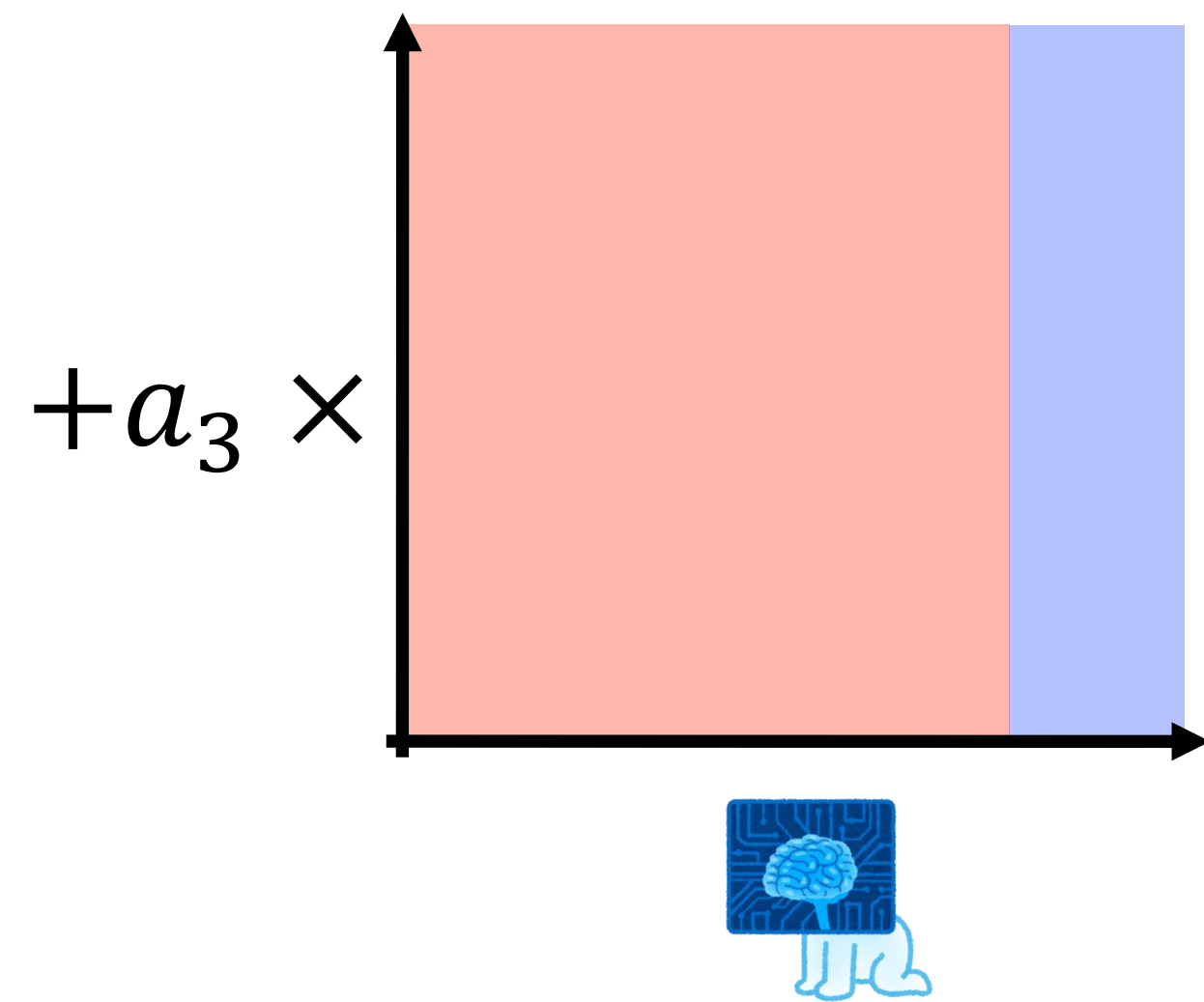
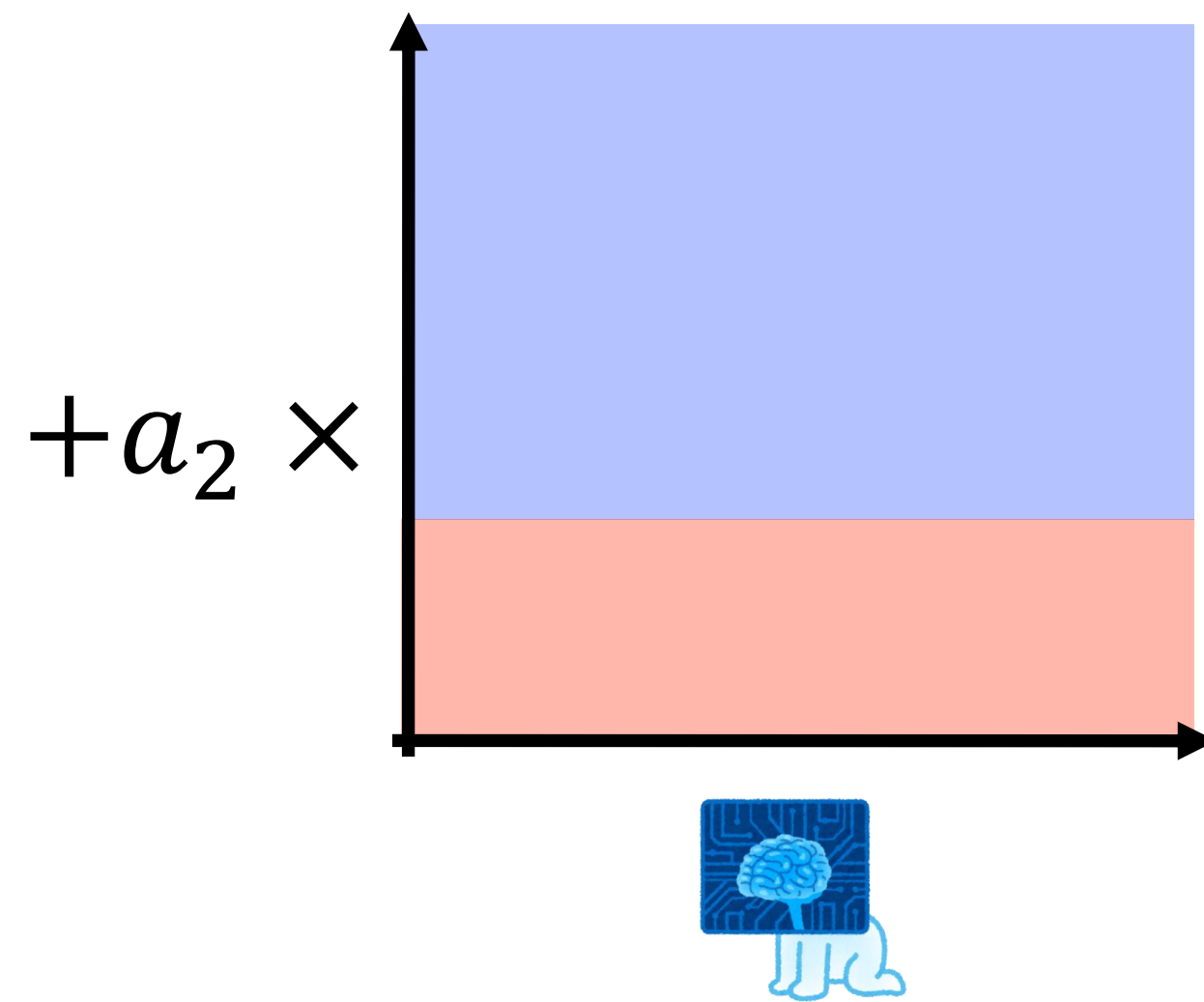
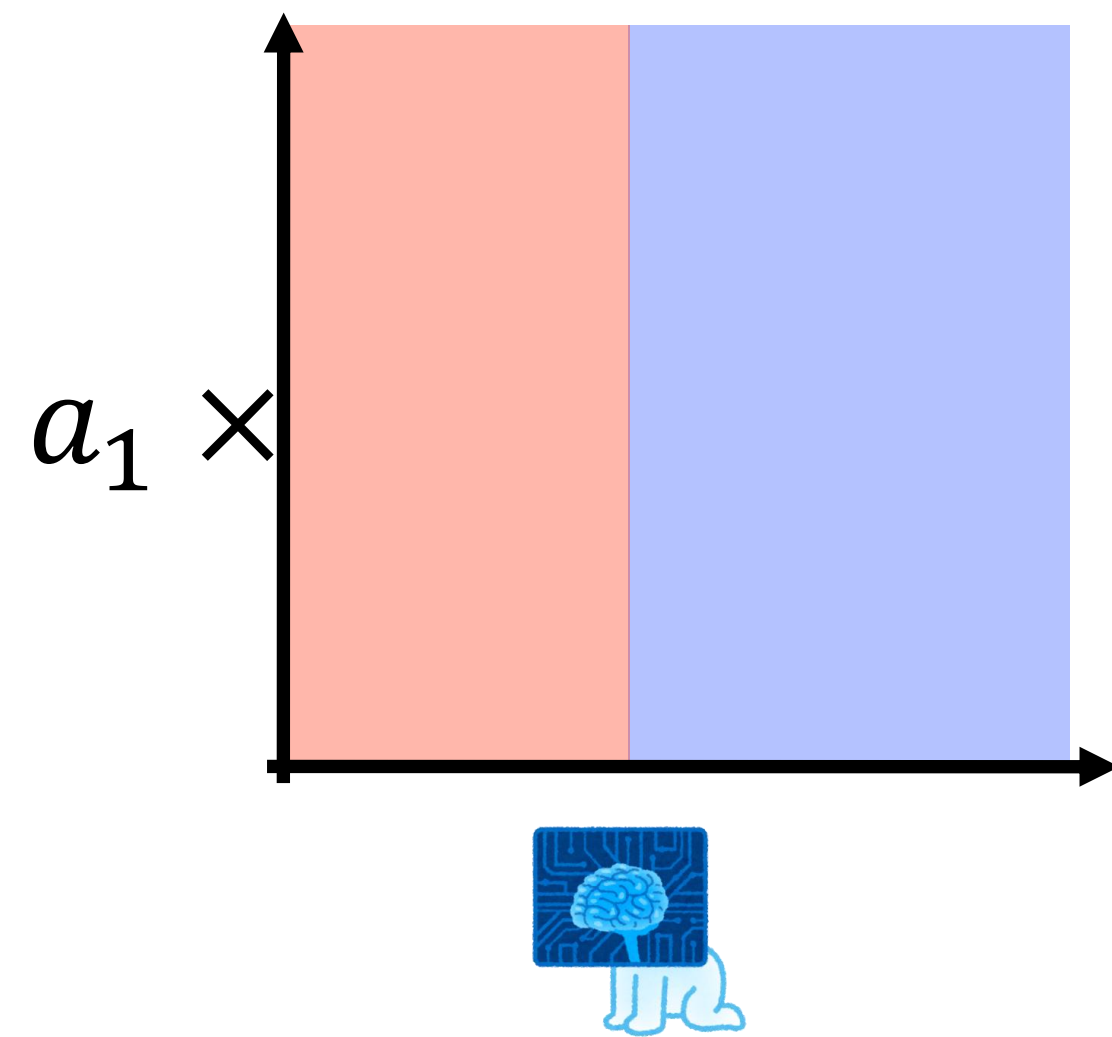
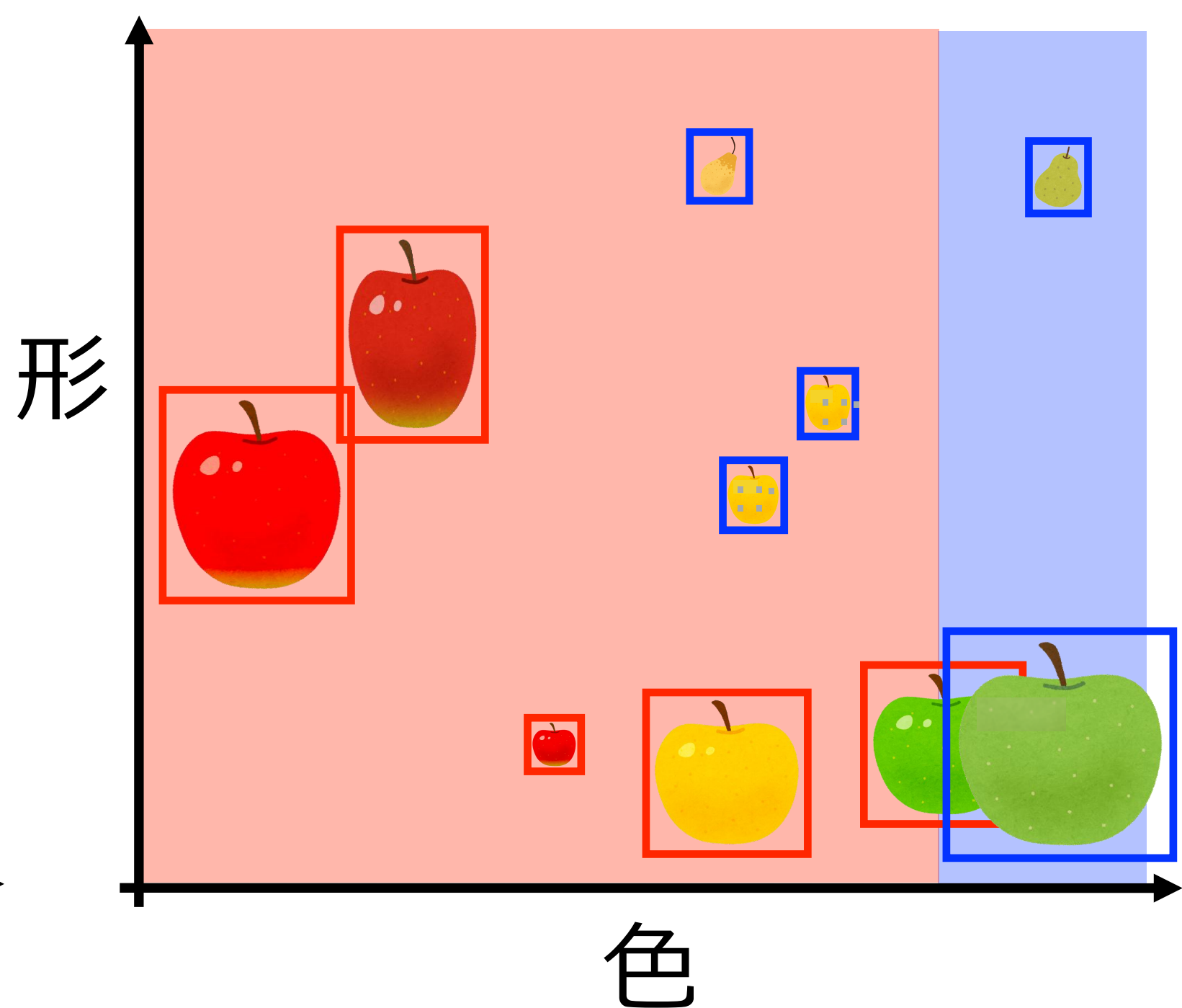
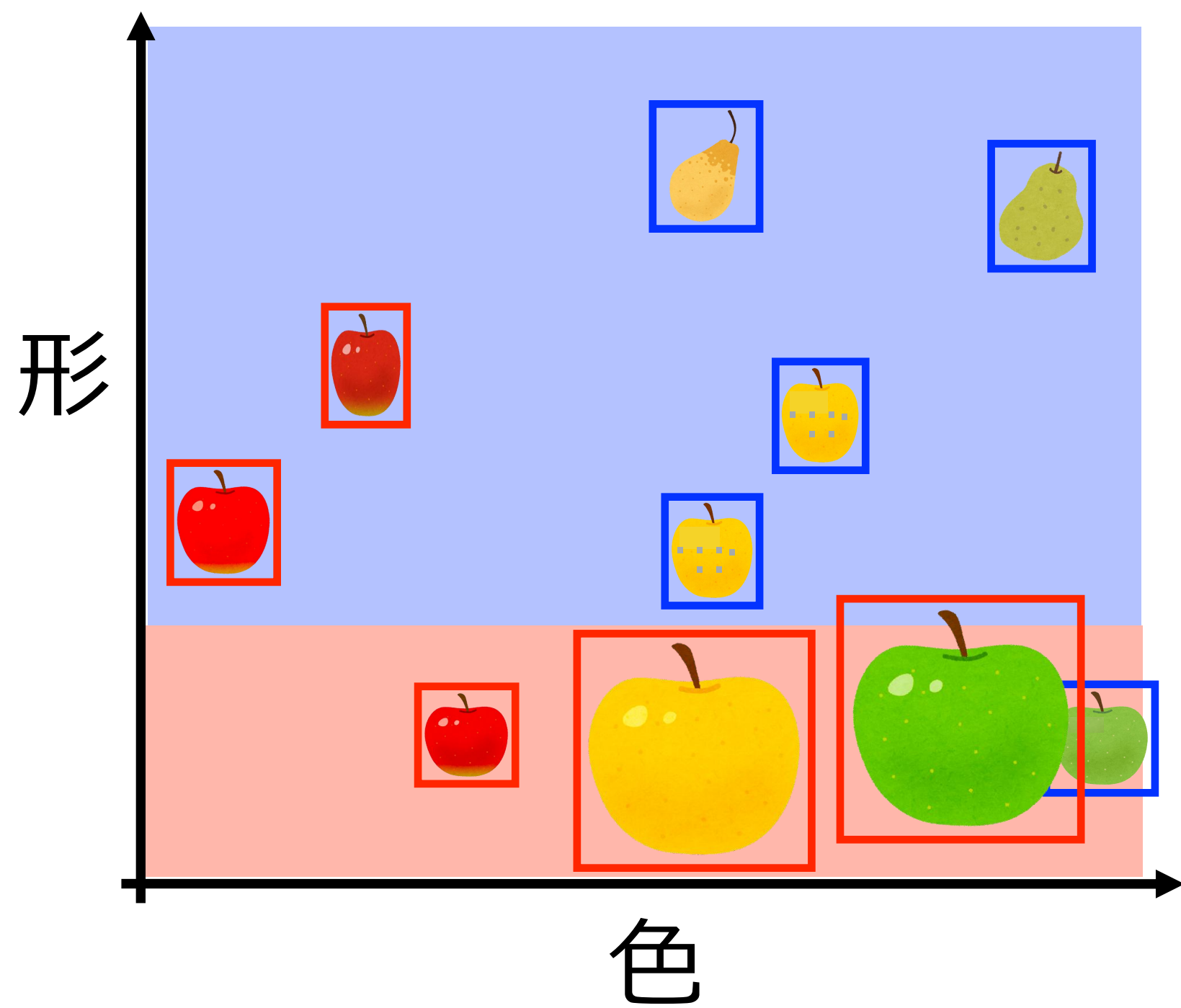
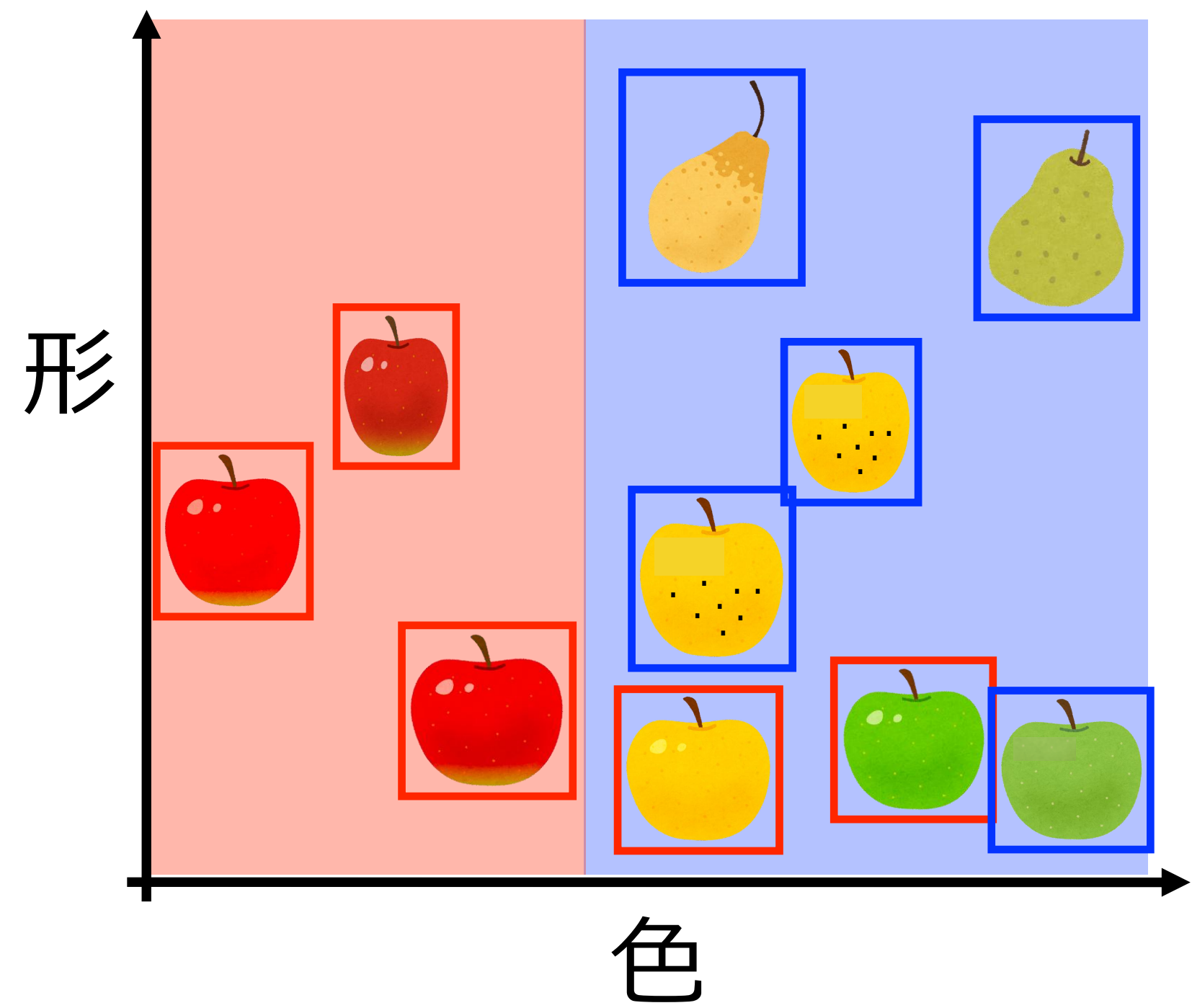






緑褐色はナシ





Boosting のメリット&デメリット

■ メリット

- ◆ 実装がとても簡単（複雑なプログラミング不要）
- ◆ 理論的な性能保証
 - T （繰り返し数）を増やすほど訓練誤差が指数的に減少
 - 非常に計算効率が良い
- ◆ 決定木（決定株）を使えば離散値も扱いやすい
- ◆ 汎用的なフレームワーク
 - 様々なタスクに拡張されている（マルチクラス分類, ランキング, 回帰）
 - 様々な発展的アルゴリズム（LPBoost [Demiriz+, 2002], Gradient Boosting [Mason, 1999] などなど）

■ デメリット

- ◆ 適当な弱仮説を用意する必要がある（なので慣習的にはよく決定株が使われる）
- ◆ ノイズに弱いことが指摘されている [Dietterich, 2000] が、実験的にはそこまでノイズに弱くないという結果もある
- ◆ 特徴は学習できないので複雑なデータの場合はあまり良い性能が出ない

XGBoost [Chen and Guestrin, 2016]

- ここ何年かはBoostingというところれ
 - ◆ 実データに対して高性能, データ分析コンペなどでもよく利用
 - ◆ 勾配Boosting [Friedman, 2001]に基づく
- 弱学習器は決定木 (CART)
- AdaBoost 同様, 前の決定木の誤差を次の決定木で修正
- 過学習抑制機能あり
 - ◆ 木の葉の数, 予測のスコア
- 並列処理が入っており高速
- 欠損値処理も自動で行い強力
- 通常のBoosting (先ほど紹介したのはAdaBoost) に比べると過学習のリスクは高い

機械学習のプロセス概要

機械学習を使った実験的解析のプロセス（超簡易版）

- たくさんの学習データ（訓練データ）を用意し、誤差を可能な限り小さくする
 - ◆ データ数が多いほど、汎化性能と経験性能のギャップが小さくなりやすい
- 得られた予測モデルを使って実際に予測してみる
 - ◆ テストデータがあれば、その精度を見て評価する

機械学習を使った解析の実践的なプロセス（詳細版）

- データの前処理（正規化&標準化, ノイズ除去, 画像リサイズ等）
- 学習器の選定
- 損失関数（目的）の決定
- ハイパーパラメータ候補の用意
- 学習データを用いて学習
 - ◆ 評価用データで「良さそうな」モデルを決定
- テストデータ **注意：ここまではテストデータのいかなる情報も使ってはいけない！**

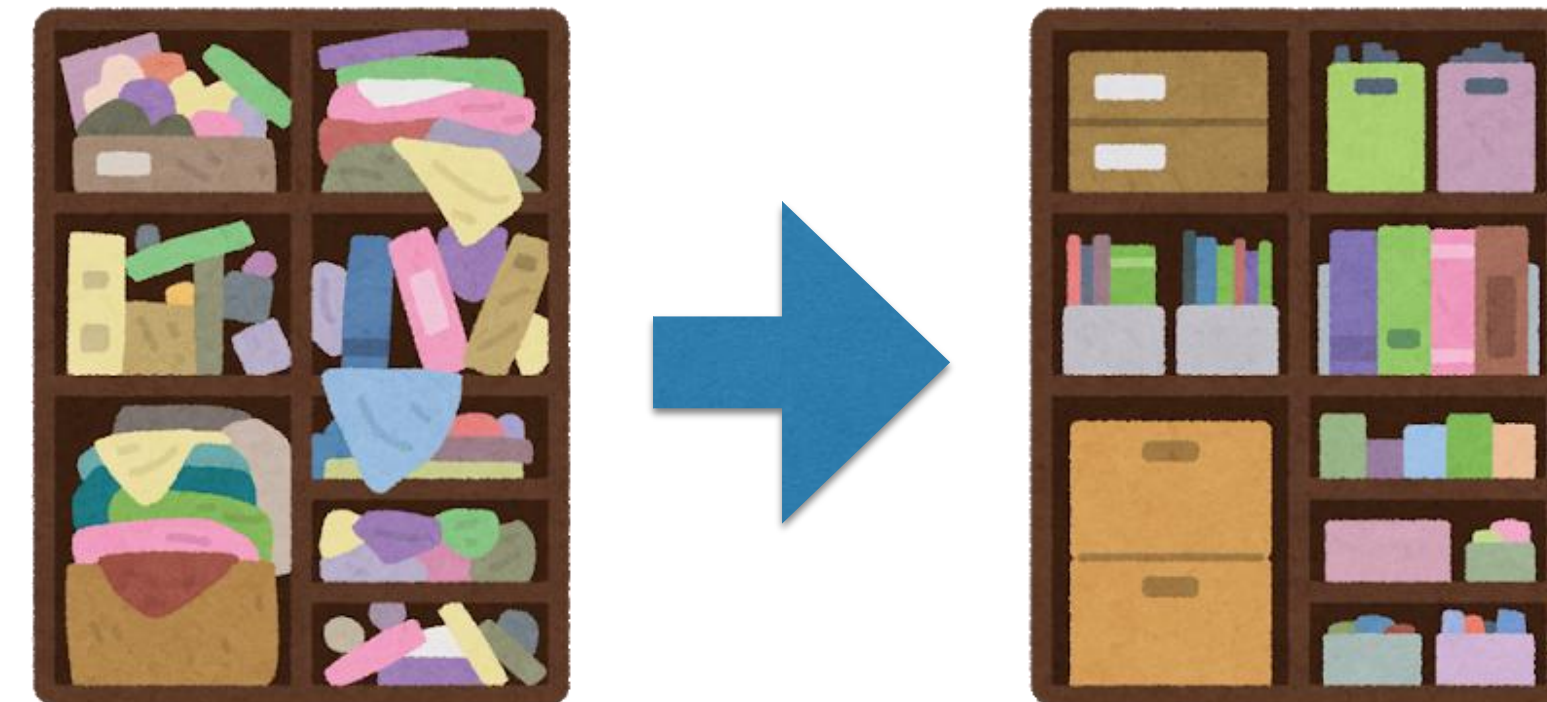
学習, 評価 (モデル選択), テスト

- 学習データ (訓練データ)
 - ◆ 学習 (予測モデルのパラメータの最適化) に用いるデータ
- 評価用 (モデル選択用) データ
 - ◆ ハイパーパラメータ (SVMで言えばC, DNNで言えば構造や最適化パラメータ, epoch 数など) の候補の数だけモデルができてしまうので, 最終モデルを決定するために用いるデータ
 - ◆ テストデータ (本番) に臨む前の「模擬試験」みたいなもの
 - ◆ 評価用データの誤差が小さいモデルでテストデータの予測に臨む
- テストデータ
 - ◆ 答えが未知のデータ (実験的には最終評価に使うので, 答えがついているが...)
 - ◆ 最終試験のようなものなので, 事前に情報を使うとチートになる

データの前処理

- 実は最終的な性能に「ものすごく」関わってくる部分です
- データをキレイに（扱いやすく）する

- ◆ ぐちゃぐちゃなのが苦手なのは機械学習も同じ
- ◆ それ以外の意味もあるが...



- ◆ 例：画像のリサイズ
（高画質の画像の場合メモリ削減のため行われる）
256 x 256 のデータを 128 x 128 にするのと、4 x 4にするのでは性能に大差が出るのは明らか

今回省略する前処理

- 欠損値補間
- 信号処理
- (高度な) 画像処理
- その他特徴抽出処理
などなど. . .

実際の「現場」でデータ解析をやっている人たちは
ここにかなり（ほとんど？）の労力を費やします. . .

データの前処理：正規化，標準化（1）

- 正規化（Normalization）, 標準化（Standardization） は原則, 必ず行う
 - ◆ 多くの学習器はデータが正規化されていることを想定している（値がバカでかくなったり, すごく小さくなったりする）
 - 例：単位が違うと, 体重50000g, 体重0.05t
 - ◆ 変数間（特徴量間）で値のスケールが大きく異なると, 重みの意味が変わってくる
 - 体重0.05, 身長1650mm
 - 不健康度 = $100 \times \text{体重} + 0.001 \times \text{身長}$
 - ↑のような場合, 「正則化」に大きな影響を与えてしまう
 - 正規化と正則化は全然意味が違うので注意

おまけ

- 平均0, 標準偏差を1に変換するのを「標準化」 (Standardization)

特定の範囲 (例えば $[0,1]$) に収めることを「正規化」 (Normalization) と呼びます

正規化, 標準化の目的

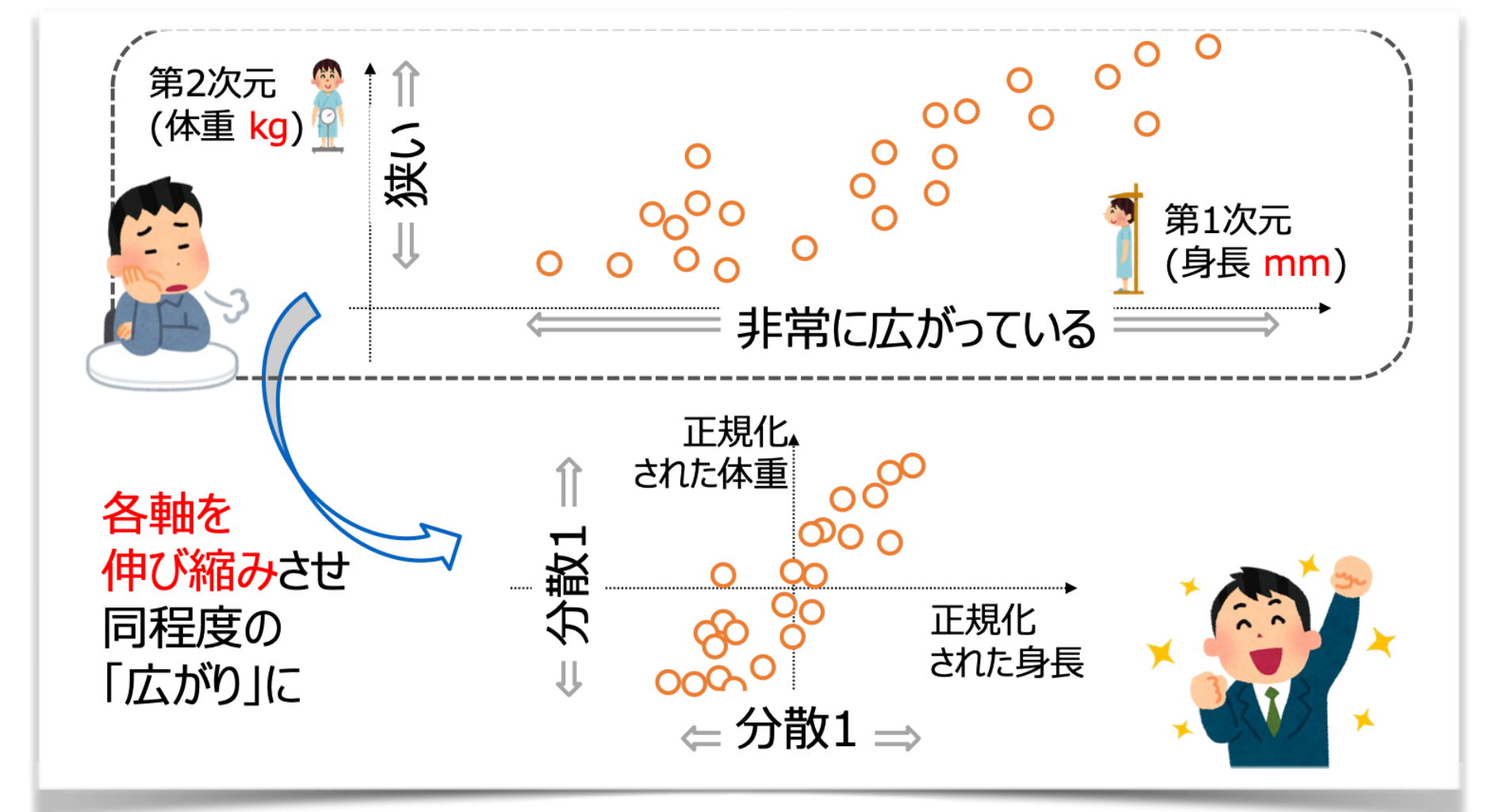
■ 各特徴量の標準化

- ◆ 特徴量のばらつき (標準的な値からどれくらい離れているか) を評価する
- ◆ 外れ値の除去 (↑に近い)
- ◆ 値のスケーリング

■ 学習理論における位置づけ

- ◆ 特徴量間のスケールを揃えることができる
- ◆ 機械学習アルゴリズムが扱いやすい

- 機械学習の実装の多くは入力の値がそこまで大きくないことを想定して作られているものが多い



データの前処理：正規化，標準化（2）

- 正規化方法には色々ある（正解はありません）
 - ◆ 画像であれば，全ピクセル値を255で割るのが一般的（0~255なので，255で割れば0~1になる）
 - ◆ 各特徴量について，最大1，最小0になるようにする
 - ◆ 各特徴量について，最大1，最小-1になるようにする
- 平均0分散1になるように標準化

標準化身長 = $(X - \text{平均身長}) / \text{身長}$ の標準偏差

Aさんの正規化身長は，
 $(175 - 173.75) / 12.5 = 0.1$

Bさんの標準化身長は -1.1

	身長	体重	健康状態
Aさん	175cm	54kg	good
Bさん	160cm	70kg	bad
Cさん	190cm	100kg	good
Dさん	170cm	45kg	bad

データの前処理：正規化，標準化（3）

- どの正規化がいいのか？
 - ◆ 値の上限，下限が決まっている場合（画像など）
 - 最小値0 (or -1), 最大値1に正規化
 - ◆ それ以外
 - 平均0, 分散1に標準化
- 残念ながら無敵の正規化，標準化方法は知られていない
 - ◆ 最小値&最大値正規化のデメリット：
 - 外れ値がある場合（例えば体重300kgの人がいる）などは良からぬ値に
 - test set では $[0,1]$ を超えてしまうかもしれない
 - ◆ 平均&分散による標準化のデメリット：
 - 特徴量が一樣でも「特徴的」にしてしまう（サイコロの目とか誕生日とか）
 - 離散値には不向き（意味が大きく変わってしまう可能性あり）

データの前処理：正規化，標準化（4）

- 注意：正規化&標準化にも，Test set のいかなる情報も使ってはいけない
 - ◆ 例えばよくある実験として，50000個のデータを持っていて，30000個を training set，10000個を validation set，10000個を test set に分割したりします
 - このとき，50000個のデータを使って正規化を行うのはNG！
 - test setの「情報」（平均や分散に関わる情報）を使ってしまっている
 - これが許されると，色々なインチキが可能になってしまいます
例：test set も含めて clustering など可能になってしまう
 - test set 内のデータの標準化は，training set の情報を使って行います
例：平均0，分散1の標準化を行っていて，
test set のデータの身長が 180 だった場合， $(180 - 173.75) / 12.5$



※画像は最大値と最小値が定まっているので，255で割る，はイカサマではない

離散値，カテゴリカル変数の扱い

■ 離散値

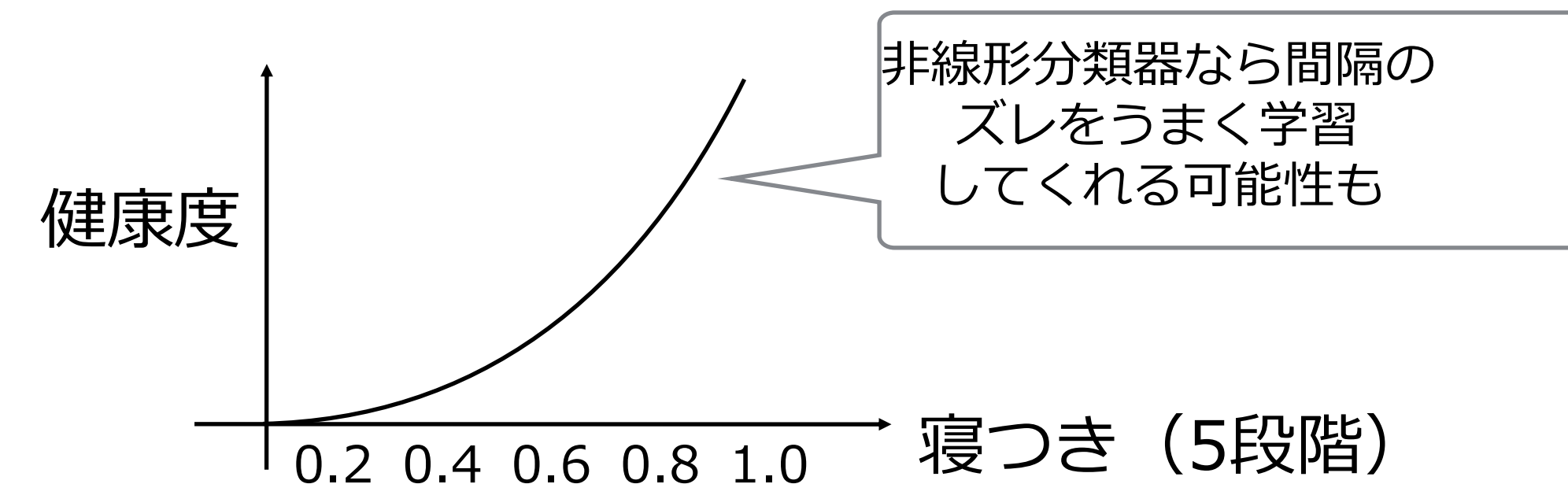
- ◆ ランク，5段階アンケート，年齢などのように実数でないもの
- ◆ 難しさ：例えば「悪い」「まあまあ悪い」「普通」「まあまあ良い」「良い」の5段階があるとき，各「間」は同じ間隔とは限らない
 - 0から1の実数にしたりすると，0～0.2，0.2～0.4の間の数値間隔が同じ意味を持つとは限らない

■ カテゴリカル変数

- ◆ 男女，出身地，所属，特性の有無など数値化が難しいもの
 - 自然言語処理における「単語」のように類似度が測れるものはベクトル化技術が進展していたりする

離散値の扱い方

- (先述のリスクを考慮したうえで) 正規化する
 - ◆ 1から5段階→0, 0.25, 0.5, 0.75, 1.0 や -1.0, -0.5, 0, 0.5, 1.0など
 - ◆ 正規化して非線形分類器を使う
 - 間隔のズレを非線形性で学習してくれることを願う
- ワンホットエンコーディング (量子化)
 - ◆ 例えば「悪い」を (1, 0, 0, 0, 0) , 「まあまあ悪い」を (0, 1, 0, 0, 0) にする
 - ◆ 離散値がカテゴリカル変数と同じ意味であれば良いがランクや段階評価などの場合, 大きさの意味を失う
- 決定木ベースの方法を使う (一番手っ取り早い)



カテゴリカル変数の扱い方

- ワンホットエンコーディング（量子化）
 - ◆ 一番ベーシックなベクトル化方法

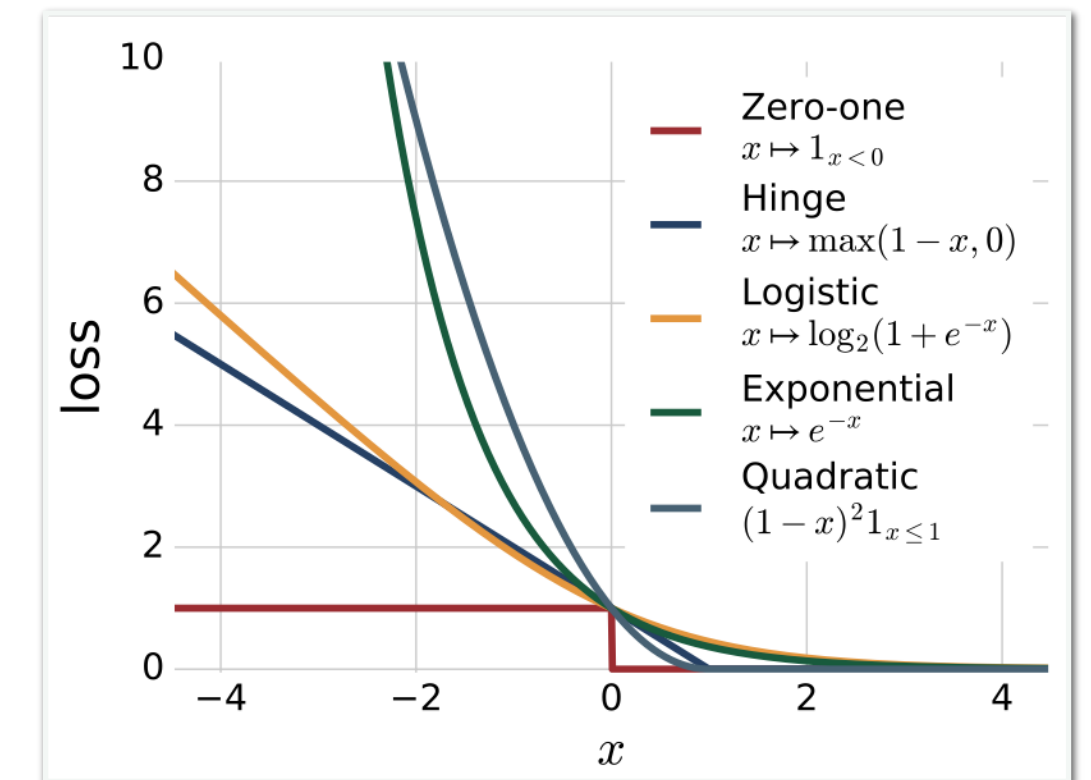
	性別	出身地	身長	体重
Aさん	男性	北海道	175cm	54kg

	男性?	女性?	沖縄生まれ?	鹿児島生まれ?	...	北海道生まれ?	身長	体重
x_1	1	0	0	0	...	1	175	54

- 決定木ベースの方法を使う（一番手っ取り早い）

損失関数

- ということが「損失」となるかを評価する関数
 - ◆ 分類だったら「間違い」, 回帰なら「ズレ」
 - ただし機械学習では最適化の都合上, 本来の損失を代替する関数が使われる
 - 例: 分類間違いをカウントするなら「間違ったら1, 正解だったら0」という損失関数が一番だが, 代替の損失関数は色々ある
 - 代替の損失関数は基本的に「多めに見積もる」もの
 - 実装のオプションで「色々損失関数あるけどこれって何だろう?」と思ったら調べてみよう
- 一番大事なのは「自分の目的に合っているか」
 - ◆ 理想は「最終評価に使う指標」 = 「損失関数」 (もしくは代替損失関数)
 - ◆ 過学習を抑制したい場合は正則化などを損失に加える



[Mohri+, 2018]

モデルの決定



ボス戦（テスト，実用）に向けた
最終パーティの決定みたいなもの

■ ハイパーパラメータの決定（モデルの決定）

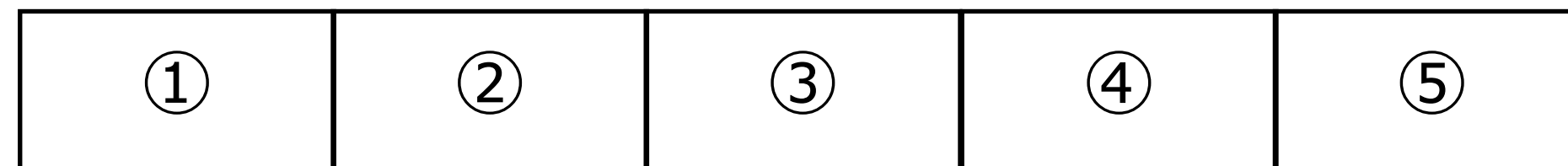
- ◆ SVM や DNNの重みベクトルは最適化アルゴリズムで決定するが、ハイパーパラメータに関しては決定が難しい
 - DNN はハイパーパラメータがめちゃ多いので、「良さげ」なハイパーパラメータ（とかDNNの構造）がいくつか提案されている
 - **デフォルト（パッケージの通常設定）が一番いいと思っではダメ！**
- ◆ 適当にいくつか用意して、「良さそうなもの」を選ぶ
よくあるのは以下の2例：
 - 評価用データに対し，最も性能が良いモデルを採用→ 評価用データに依存する可能性がある
 - Cross validation（交差検証）を行い，最も平均性能が良いモデルを採用
→ 固定の評価用データよりも安定した評価が可能だが，計算コストが大きい

(k-fold) Cross validation

■ 手順

- ◆ オリジナルの学習データを k 個に分割する
- ◆ $(k-1)$ 個のブロックを改めて学習データとし、残りの1ブロックを評価用データにする
- ◆ k 回分行う

■ 例 : 5-fold cross validation



Train: ①, ②, ③, ④ validation: ⑤

Train: ①, ②, ③, ⑤ validation: ④

Train: ②, ③, ④, ⋮ ⑤ validation: ①

} 5回での平均 validation error を計算

Cross validationに関するその他

- Test samples が明示的に与えられていないとき
（例：50000個の答え付きデータを自分で持っている）は、汎化性能を Cross validation で評価することもある
 - ◆ テストデータを k 回変えて評価する
 - ◆ この場合もテストデータを含めて正規化・標準化しないように注意
- k-fold の k は大きいほど信頼性が増す
（汎化誤差に近くなる）が、計算コストの兼ね合いで決める
 - ◆ k がサンプル数の場合、Leave-one-out と呼ばれる
- Cross validation は python で簡単に行える

グリッドサーチ

- ハイパーパラメータの組み合わせを網羅的に調べる方法
 - ◆ 例：ランダムフォレストの木の数と木の深さを色々変えて良いものを探す
木の数の候補：50, 100, 200, 300 木の深さ 5, 10, 15

	50	100	200	300
5	(50, 5)	(100, 5)	(200, 5)	(300, 5)
10	(50, 10)	(100, 10)	(200, 10)	(300, 10)
15	(50, 15)	(100, 15)	(200, 15)	(300, 15)

sklearn.model_selectionのGridSearchCVを使うと簡単に実装可能
(交差検定で全ハイパーパラメータ候補の組み合わせから一番良いものを見つける)

演習

はじめに

- 機械学習のコードは複雑になりがち
- ChatGPTやGeminiなどの生成AIをうまく活用しよう
 - ◆ ただし動作確認は必須（←超重要）
 - バグがないからと言って「思い通り」の動作とは限らないので本当に要注意
 - ◆ プロンプト（命令文）を可能な限り細かく書くこと
 - 指定されていないことは勝手に解釈されて好き放題書かれる可能性もあり
 - ◆ 各行にコメントアウトで何をしようとしているのか説明をするように指定するのも良い
- 今回の演習用のコードもChatGPTやGoogle Colabの生成AIを活用して作成しています

SVMの実装

- 正規化 or 標準化
- 損失関数の選択
 - ◆ ペナルティは通常 $\sum_{i=1}^m \xi_i$ だが $\sum_{i=1}^m \xi_i^2$ とすることもある (squared hinge などと呼ぶ)
 - ◆ SVMはマージン最大化を行うので勝手に正則化項が入っているが種類はある (後述)
- C の候補を用意
- モデル選択 (交差検定)
 - ◆ 学習データの一部を評価用データに設定し, 学習 & 評価を数度行う
- テストデータを使って汎化性能を評価

人工データに対する実験

- 5-fold 交差検定で最適な C を決定
- 訓練誤差とテスト誤差を表示
- 分類領域を可視化
 - ◆ 今回使うのは2次元人工データなので可視化可能

実装のポイント

- SVMのライブラリ & モデル選択のライブラリをインポート

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, train_test_split
```

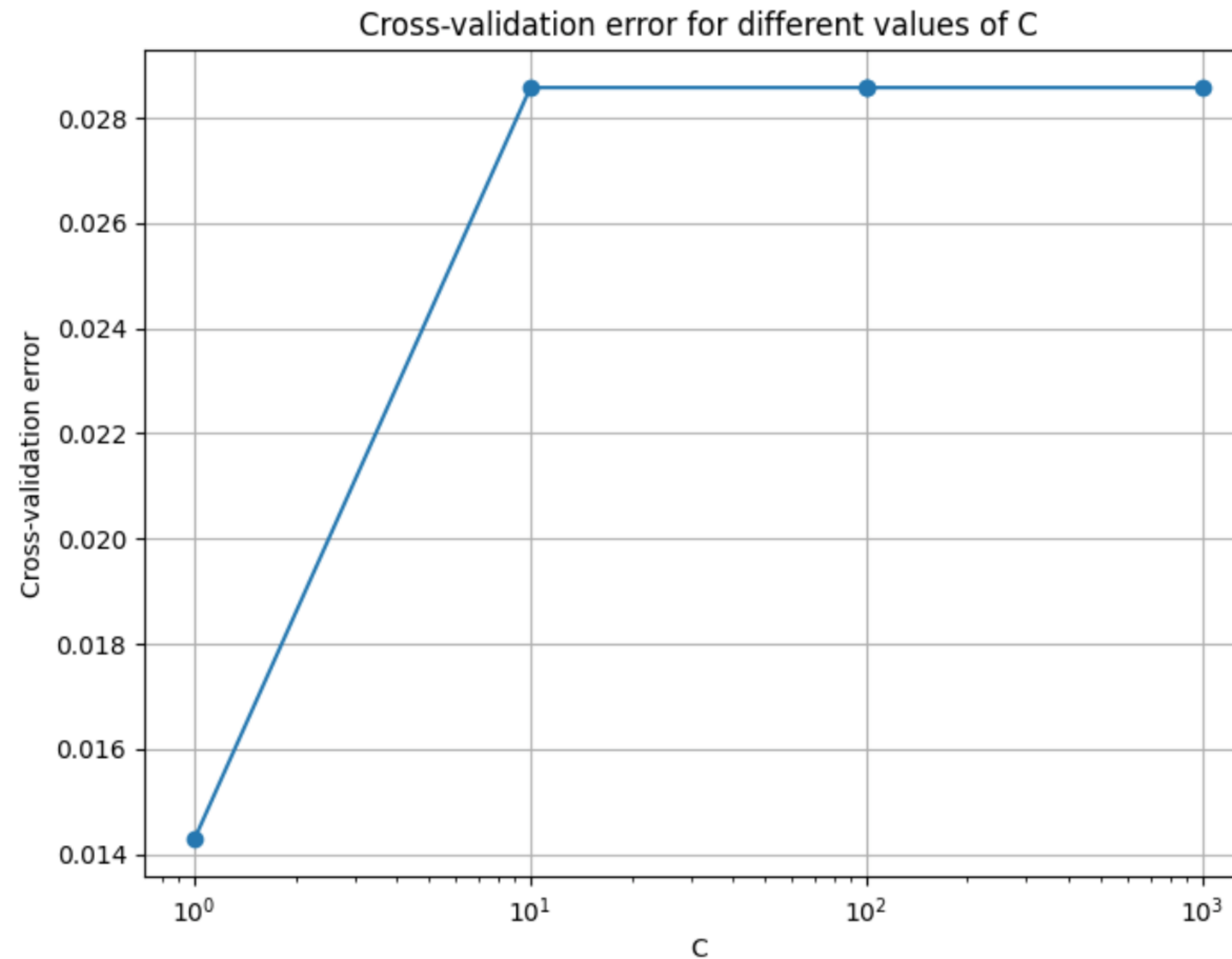
- 交差検定で「良い C 」を見つける
 - ◆ `kernel='linear'` にすることで線形分類器を学習

```
# 交差検定で最適なCを選択
for C in C_candidates:
    svm = SVC(kernel='linear', C=C)
    scores = cross_val_score(svm, X_train, y_train, cv=5) # 5分割交差検定
    cv_errors.append(1 - np.mean(scores)) # エラー率を格納
    print(f'C={C}, cross validation error: {1 - np.mean(scores)}')
```

- 「良い C 」でモデルを学習する

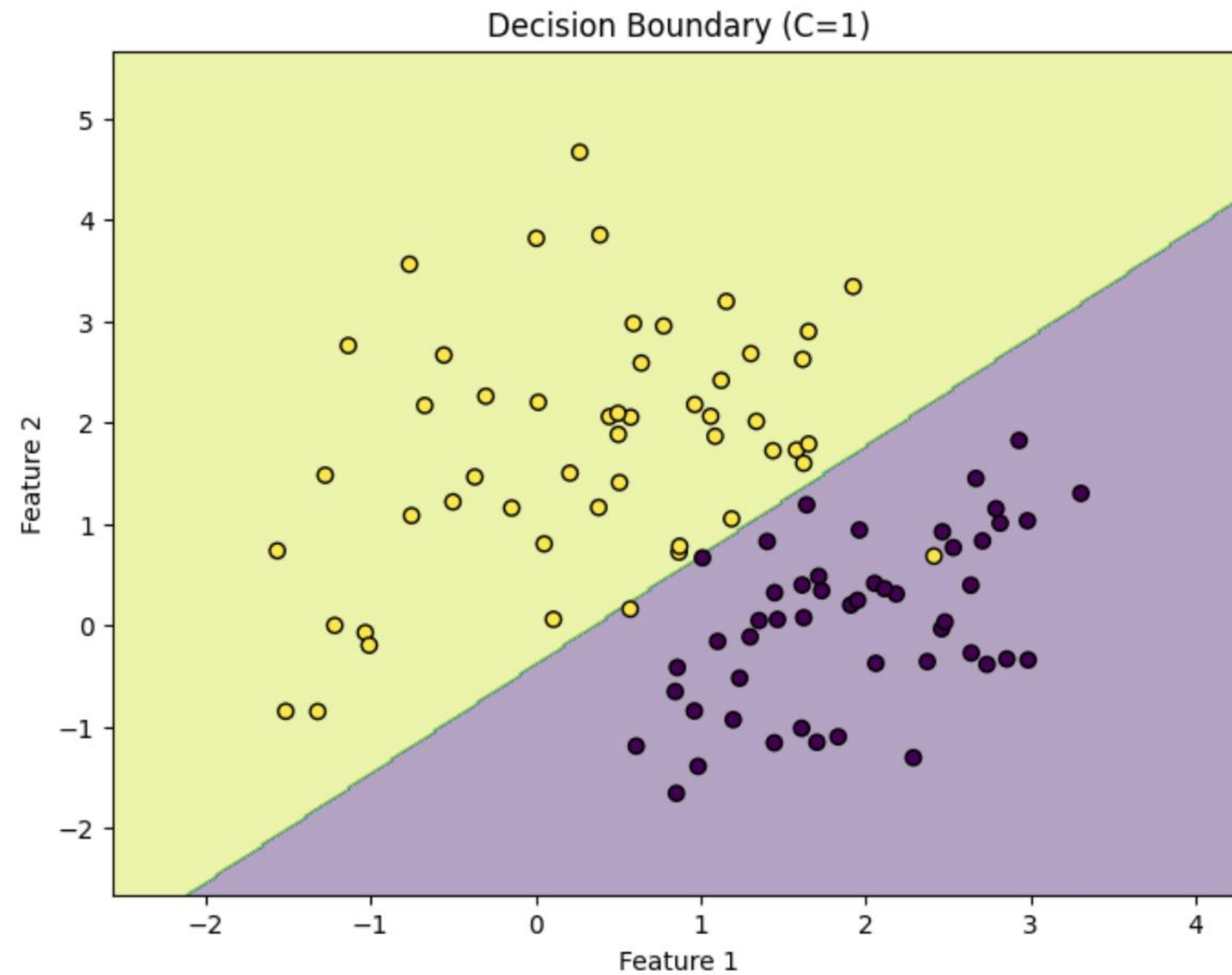
```
# 最適なCでSVMを学習
best_svm = SVC(kernel='linear', C=best_C)
best_svm.fit(X_train, y_train)
```

良い C の選択



このような結果が出た場合はより小さい C を試す価値あり

領域の可視化



訓練誤差: 0.014285714285714235
テスト誤差: 0.033333333333333326

実データに対する実験

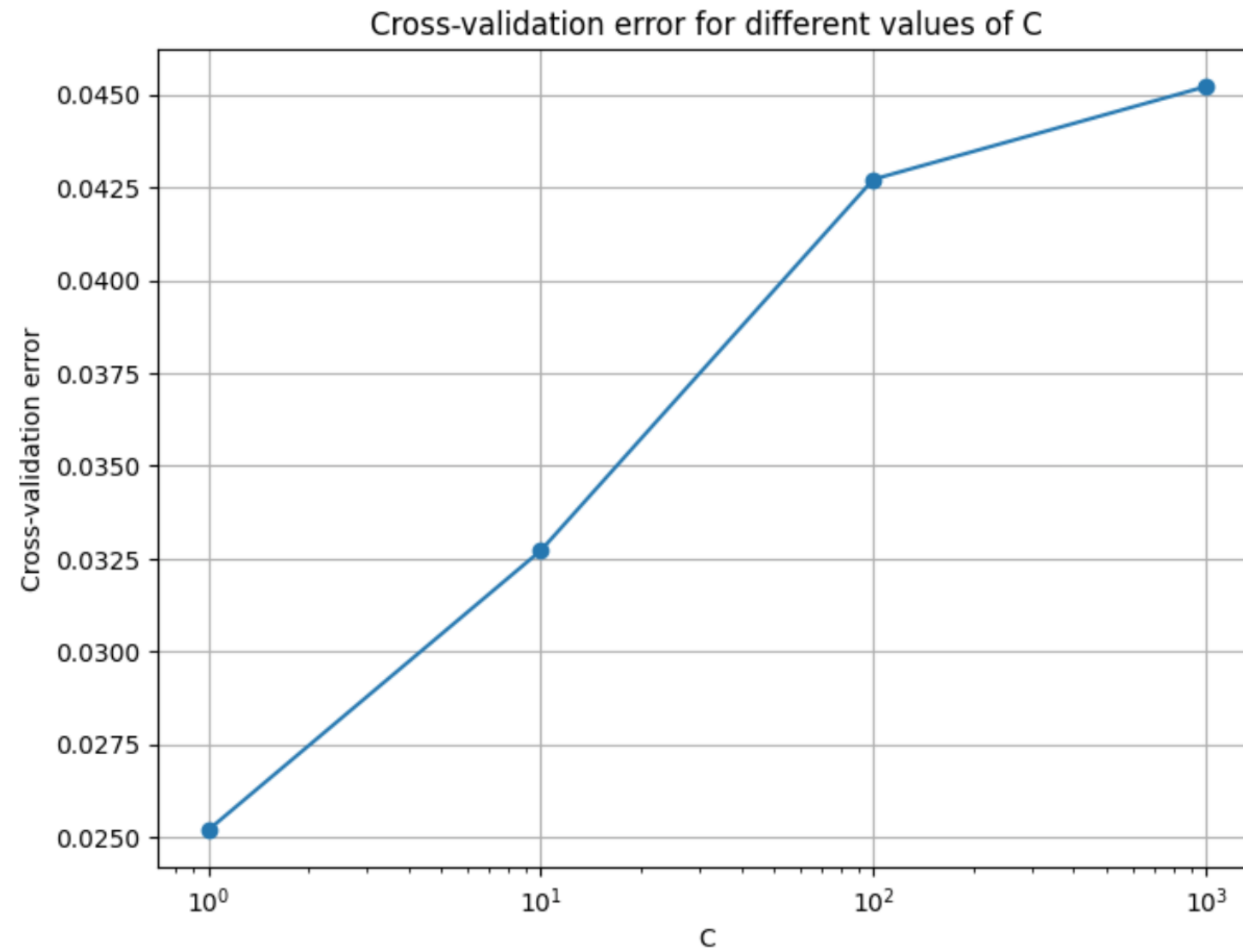
- breast-cancer データ（乳がんの診断データ）
- データ数：569
- 特徴量：30の実数値（標準化や正規化は未実施）
- ラベル：0が良性，1が悪性
 - ◆ SVM の場合はラベルを $\{-1, +1\}$ に修正する

実装のポイント

- 学習データ, テストデータの標準化を行う場合, 学習データの平均や分散を使って行う
 - ◆ 学習データとテストデータを分割する前に標準化しないこと!

```
# データの標準化
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

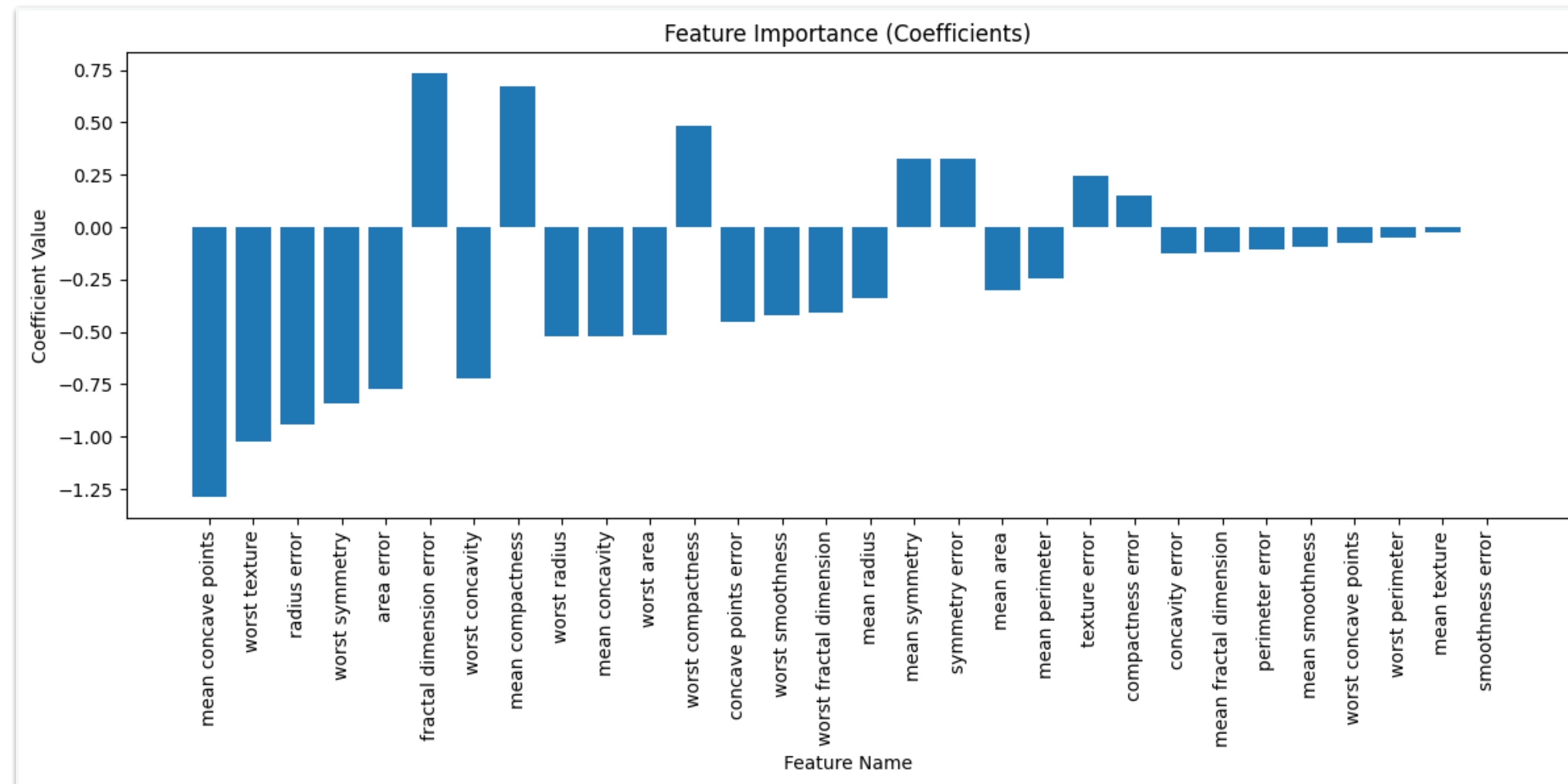
結果



最適なC: 1
訓練誤差: 0.01005025125628145
テスト誤差: 0.023391812865497075

重みベクトル（超平面）の観察

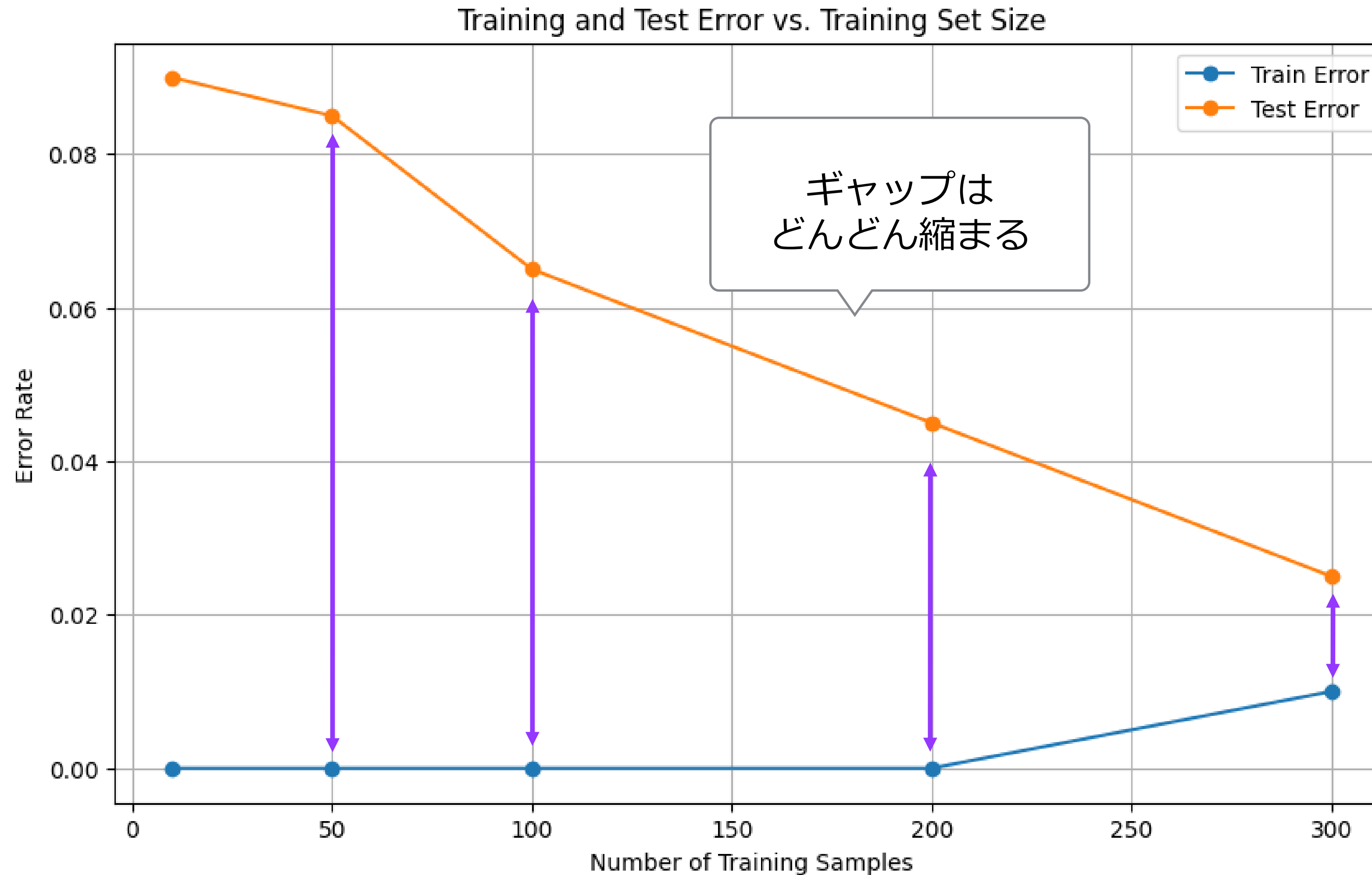
- 重みベクトルの絶対値が大きい = 分類への寄与度が大きい
 - ◆ データが正規化, 標準化されている場合に限る
- 正の重みは正例に寄与, 負の重みは負例に寄与



訓練性能と汎化性能の観察

- データ数を増やすと汎化性能は良化するか？
- cancer データで学習データ数を
 - ◆ 50, 100, 200, 300 と変化させて実験
 - ◆ テストデータは200で固定

結果



このような結果観察から「よし、もっとデータを集めてみよう」と判断できる

余裕がある人向け：スパース化

- スパースって何？
 - ◆ 疎（そ）、スッカスカ
- SVMのスパース化とは？
 - ◆ 超平面 w の要素がゼロばかり

正則化項 $\|w\|^2$ のところを
1ノルム正則化にすることで実現可能

$$\|w\|_1 = |w_1| + \dots + |w_N|$$

例：異性の好みのタイプを学習

スパースでない超平面：

$$\text{好きなタイプか?} = (0.3, 0.2, 0.2, 0.5, \dots, 0.15) \cdot (\text{年収}, \text{体型}, \text{学歴}, \text{顔}, \dots, \text{性格}) + 2.5$$

w x b

スパースな超平面：

$$\text{好きなタイプか?} = (0, 0, 0, 0.8, \dots, 0.2) \cdot (\text{年収}, \text{体型}, \text{学歴}, \text{顔}, \dots, \text{性格}) + 2.5$$

w x b

解釈がしやすい

1-norm 正則化 SVM

- LinearSVC をインポート

```
from sklearn.svm import LinearSVC
```

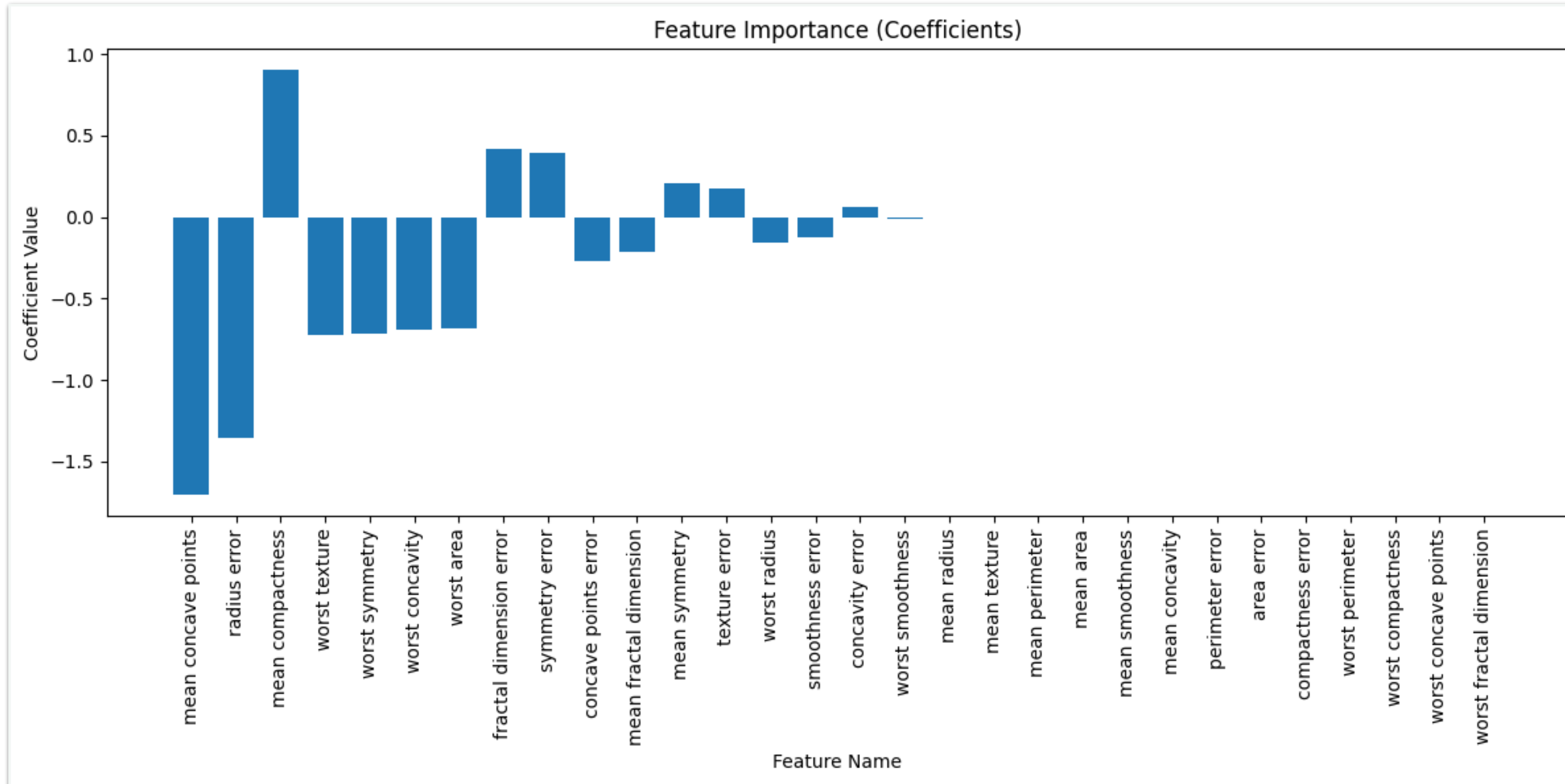
- `penalty='l1'` に設定

```
svm = LinearSVC(penalty='l1', loss='squared_hinge', dual=False, C=C, random_state=42)
```

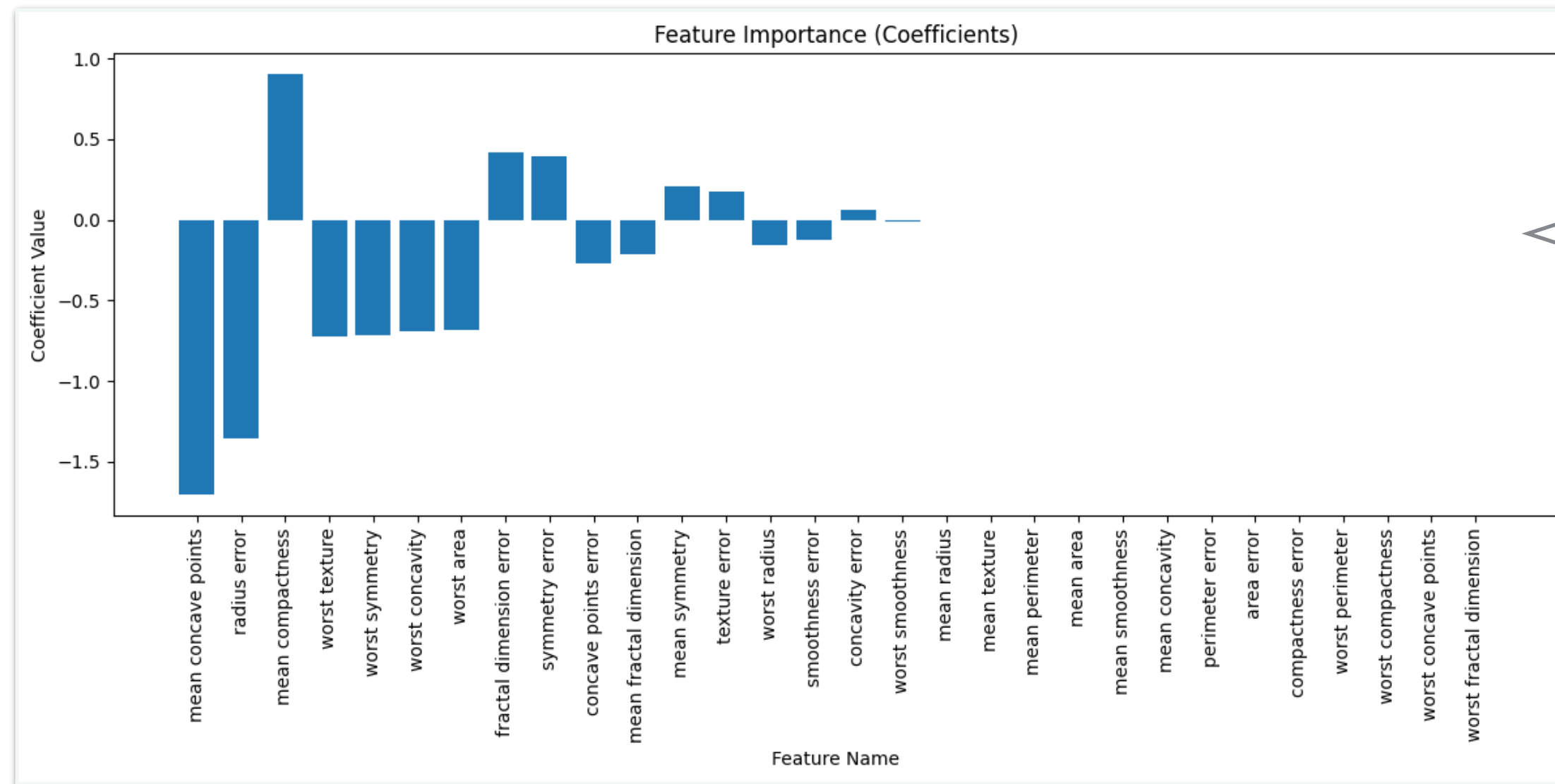
- ◆ 実装の都合上, 1-norm正則化の場合は squared hingeのみに対応なので注意

結果

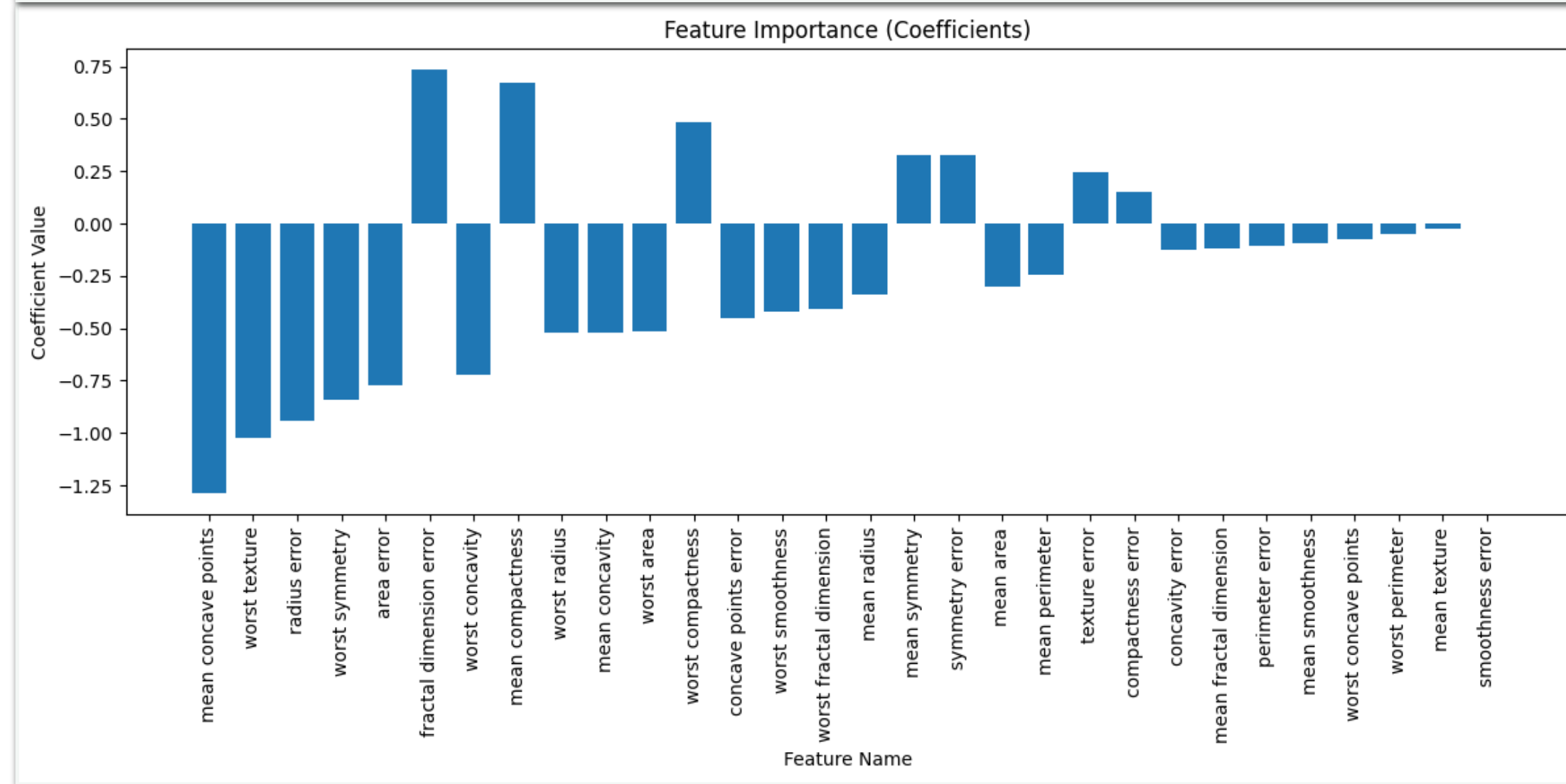
Training error: 0.01005025125628145
Testing error: 0.0292397660818714



2ノルム正則化との比較



よりスパース！
- 解釈性の向上
- 過学習の抑制
の効果あり



SVM スパース化のメリット・デメリット

■ メリット

- ◆ w の非ゼロ要素が減るため、より解釈がしやすい
- ◆ 高次元データにも強い！
 - 高次元データの場合過学習しやすいが、抑制効果あり

■ デメリット

- ◆ 2ノルム正則化よりもさらに単純なモデルになるため、訓練誤差は大きくなりやすい
- ◆ 通常のSVMと異なり非線形拡張ができない

多クラス学習

- 多クラスデータを扱うことが可能
 - ◆ SVCパッケージのデフォルトは簡素なアルゴリズムで、単純な多数決を行う（色々オプションあり）
 - 3クラスの場合：1vs2, 2vs3, 1vs3を行って、最も得票数が高いもの
 - ◆ LinearSVCというパッケージを使うと One-vs-Restになる
 - 中では1vsその他, 2vsその他, というのを何度も解いている
- 今回はLinearSVCを採用して実験
- 重みベクトルはクラスごとにできるのでそれぞれ観察できる
- Wineデータを使って実験してみよう

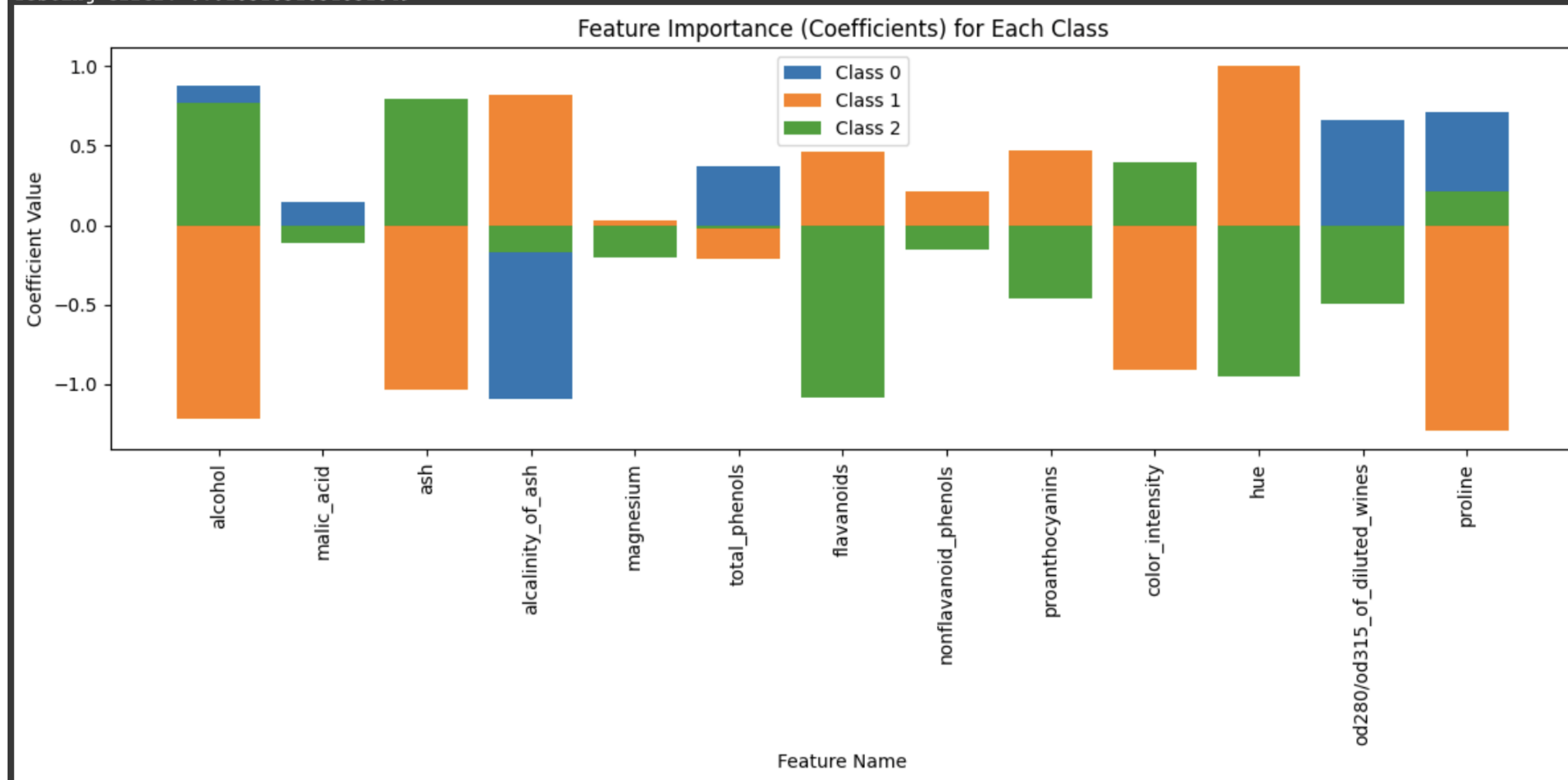
結果

```
C = 1, Cross-validation error: 0.0160000000000000014  
C = 10, Cross-validation error: 0.0160000000000000014  
C = 100, Cross-validation error: 0.0080000000000000007  
C = 1000, Cross-validation error: 0.0160000000000000014
```

Best C: 100

Training error: 0.0

Testing error: 0.01851851851851849



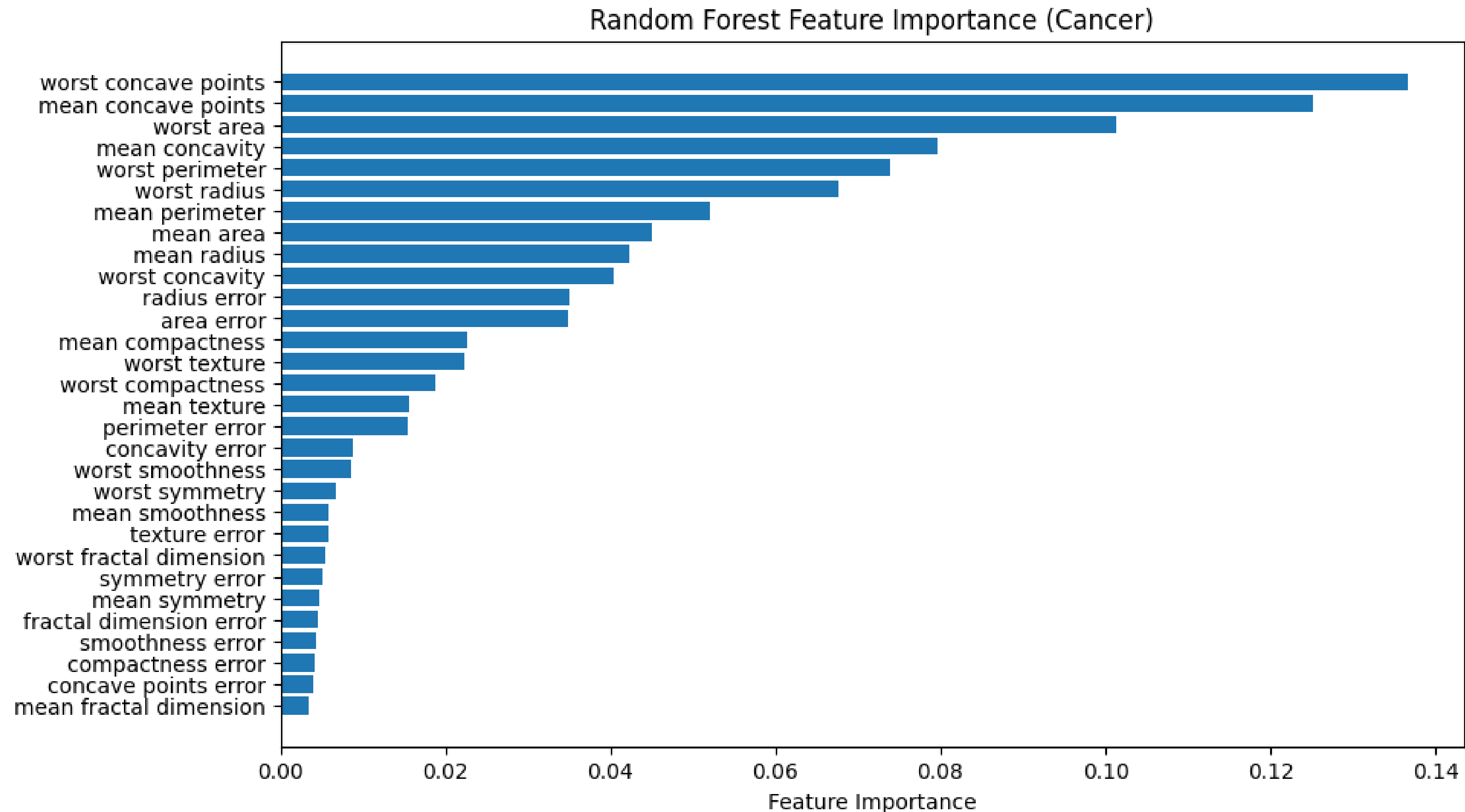
Random Forest と Boosting (AdaBoost) の実験

- 人工データと cancer データで実験
- ハイパーパラメータ
 - ◆ RF: 決定木の数, 木の最大深さ
 - ◆ AdaBoost: 繰り返し数 (決定株の数)
- 各変数の重要度を出力

Random Forest の結果

◆ Random Forest (Synthetic Data)
Train Acc: 0.9988, Test Acc: 0.9300, Time: 0.51s

◆ Random Forest (Cancer Data)
Train Acc: 1.0000, Test Acc: 0.9649, Time: 0.60s

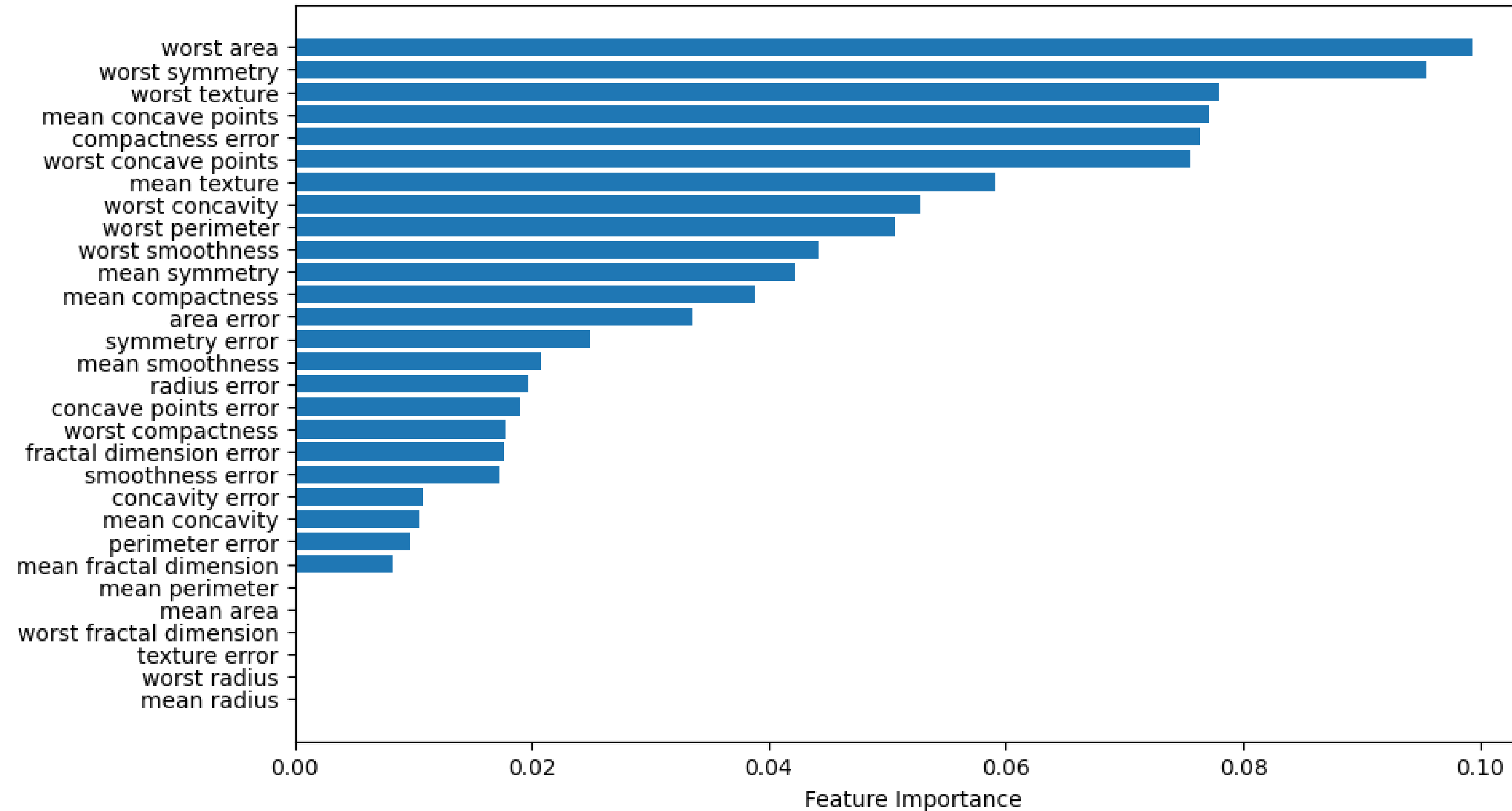


AdaBoost の結果

```
◆ AdaBoost (Synthetic Data)
Train Acc: 0.9287, Test Acc: 0.9350, Time: 0.34s

◆ AdaBoost (Cancer Data)
Train Acc: 1.0000, Test Acc: 0.9737, Time: 0.32s
```

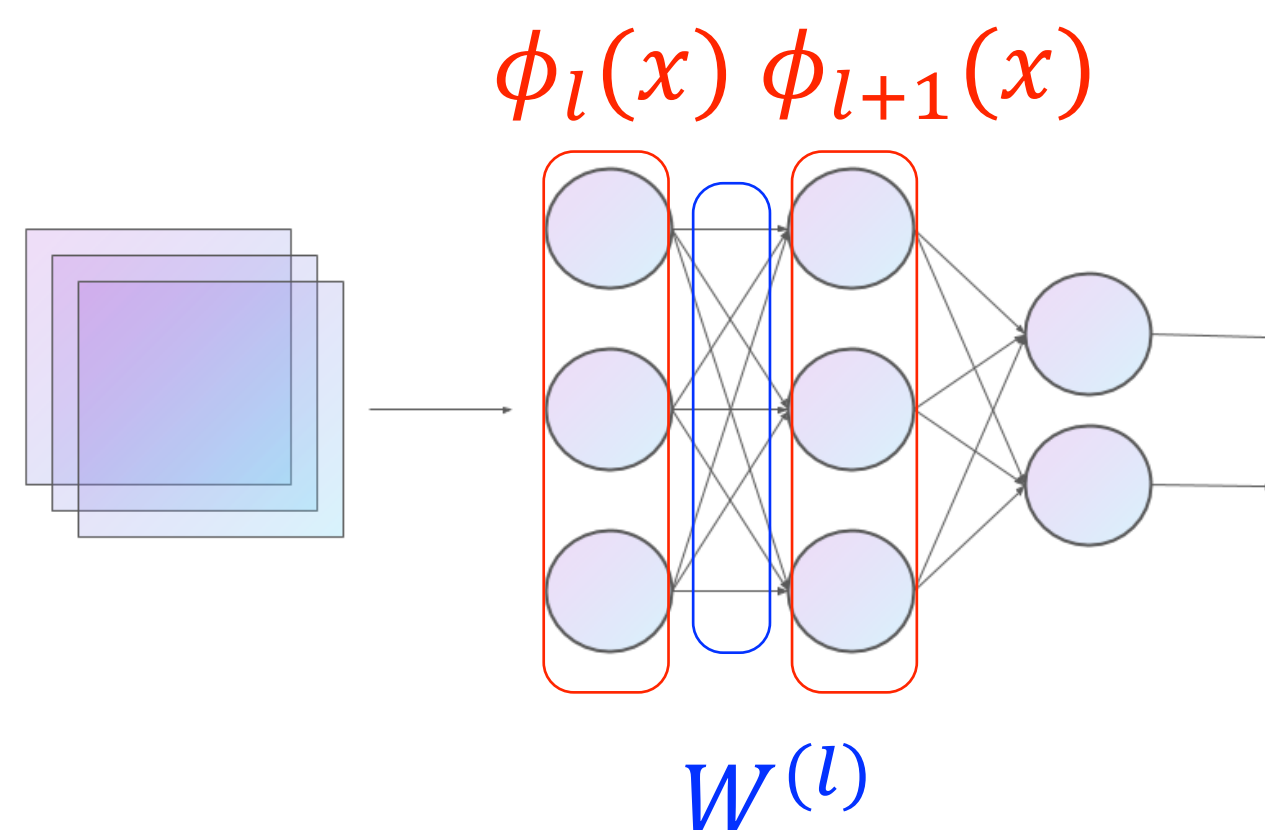
AdaBoost Feature Importance (Cancer)



ニューラルネットワーク

ニューラルネットワーク

- 入力に対してアフィン変換（線形変換＋平行移動）と非線形変換（活性化関数による変換）を複数組み合わせることで予測を行う
- 画像や自然言語処理など複雑な認識タスクに非常に有効



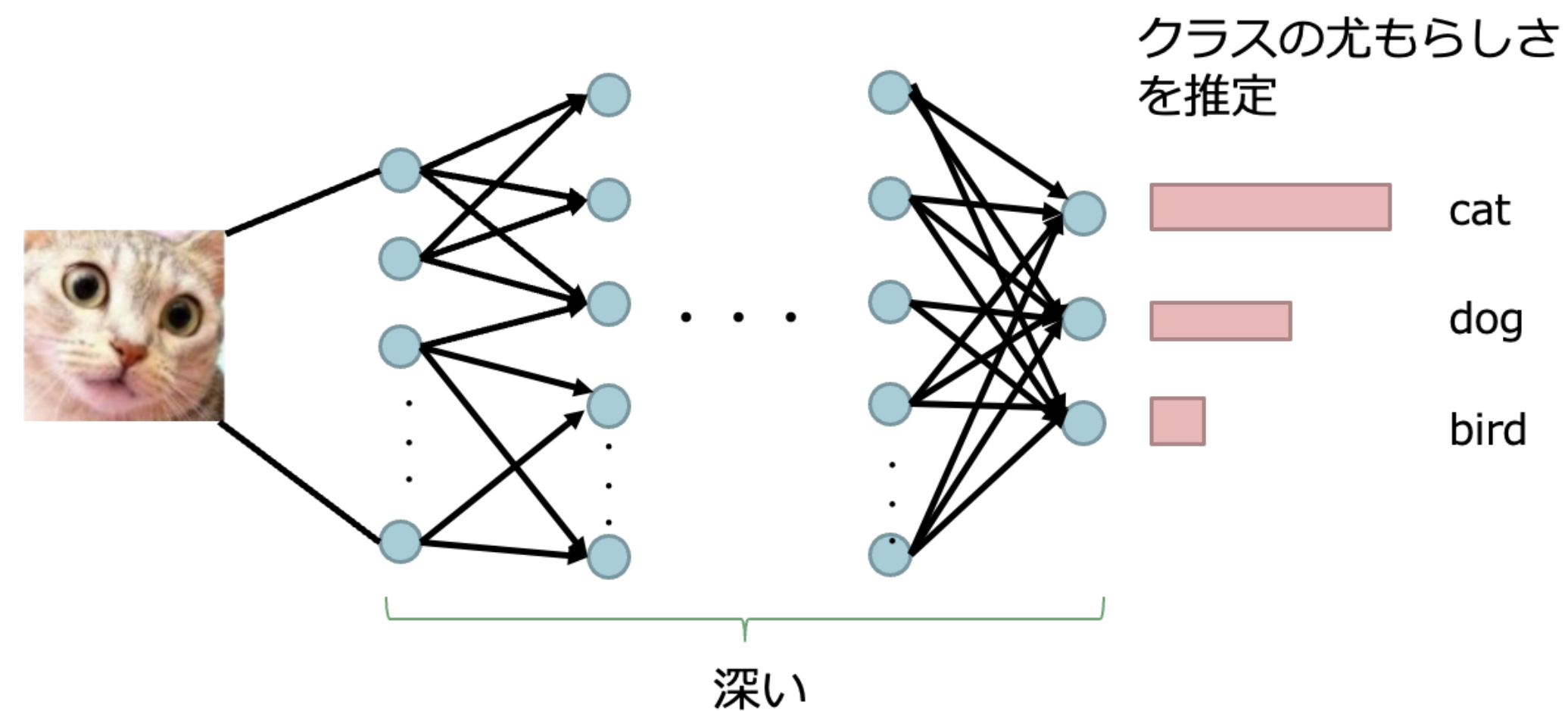
層 l にノードが N_l 個
各層 l で非線形写像 ϕ_l を行う

バイアス（定数項）の
存在は基本的に無視して
問題ありません

$$\phi_{l+1}(x) = \sigma(W^{(l)} \phi_l(x) + b^{(l)})$$

ニューラルネットワークの特徴

- 特徴も学習可能
 - ◆ 非線形変換を繰り返すことで複雑な特徴写像も表現可能

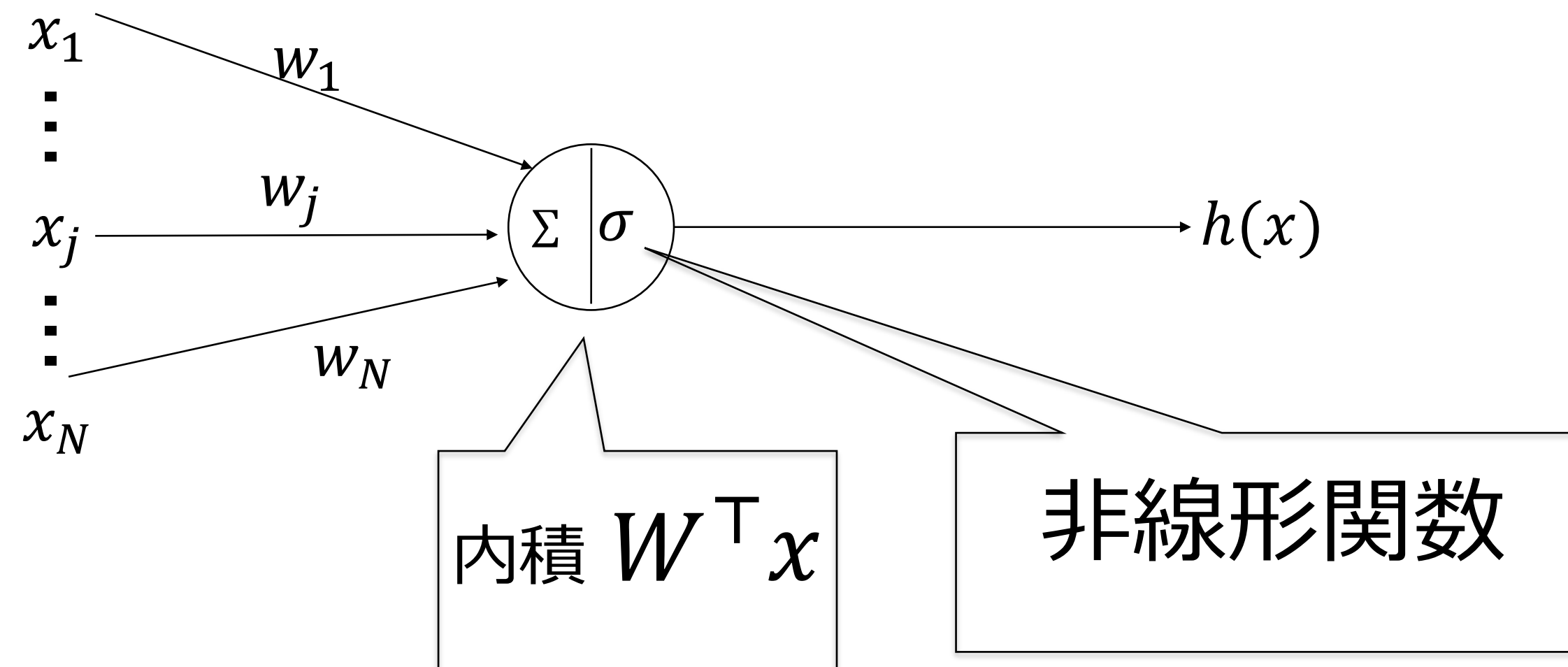


直感的な理解に向けて

- 1層のネットワークを考えてみよう

入力 x : N 次元ベクトル

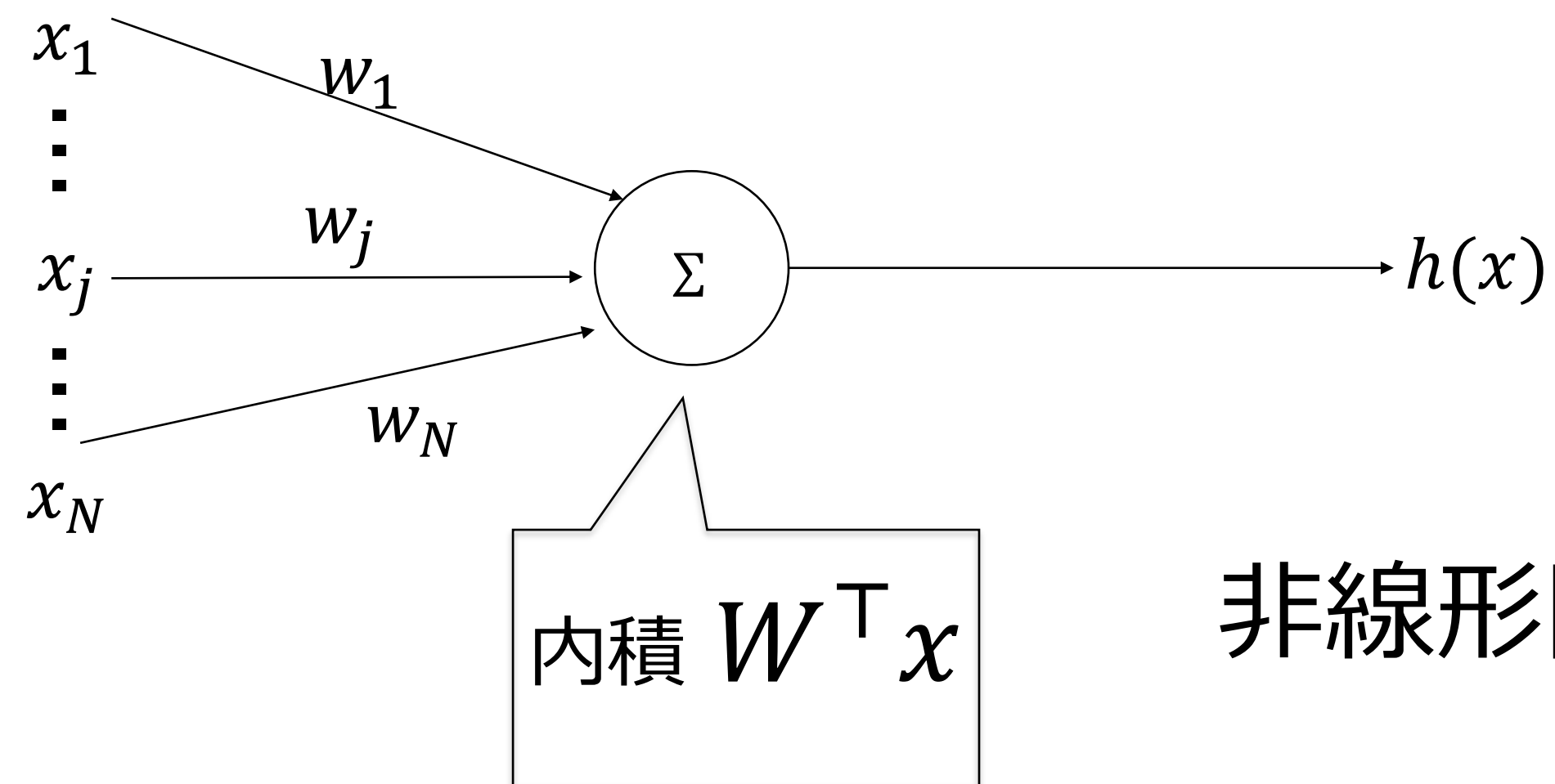
$$h(x) = \sigma(W^T x)$$



直感的な理解に向けて

- 1層のネットワークを考えてみよう
- 非線形関数もなくして考えてみよう

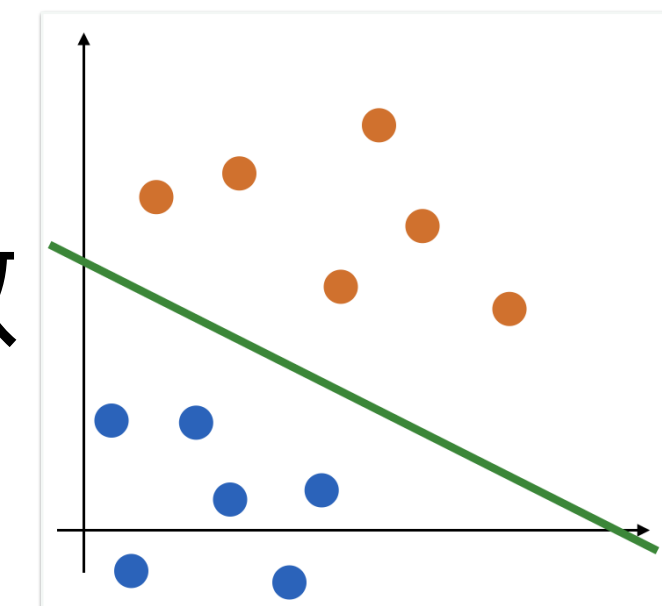
入力 x : N 次元ベクトル



$$h(x) = W^T x$$

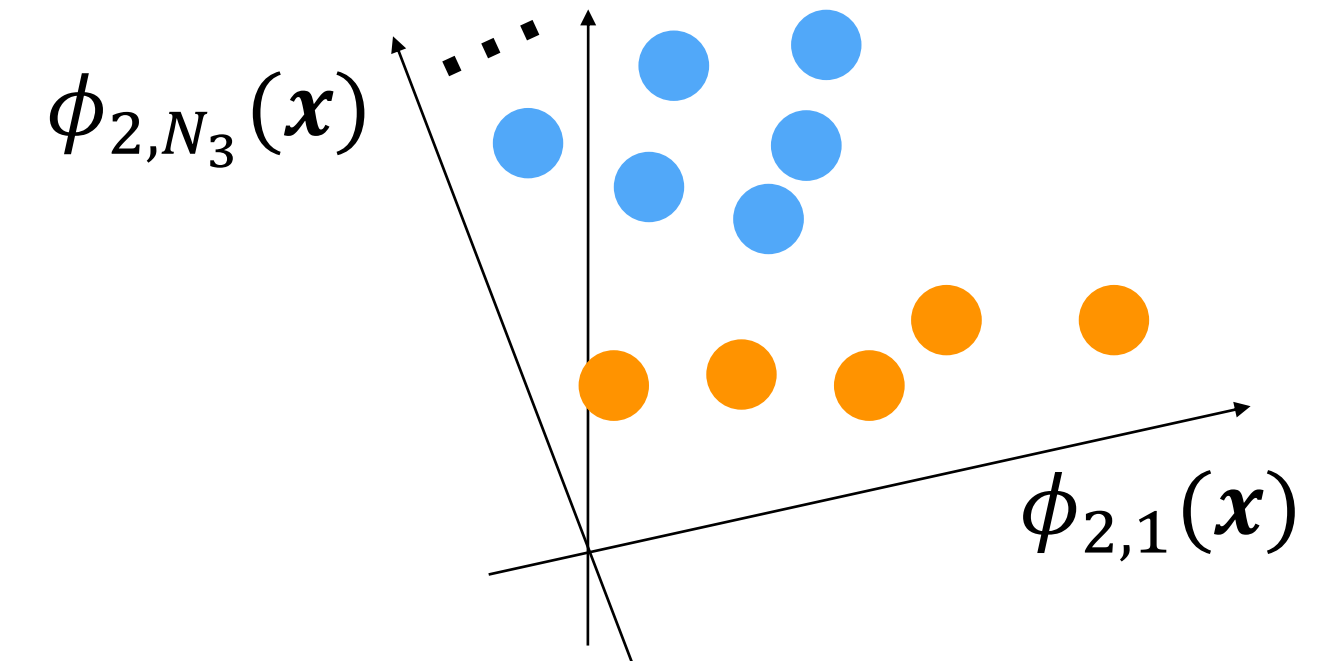
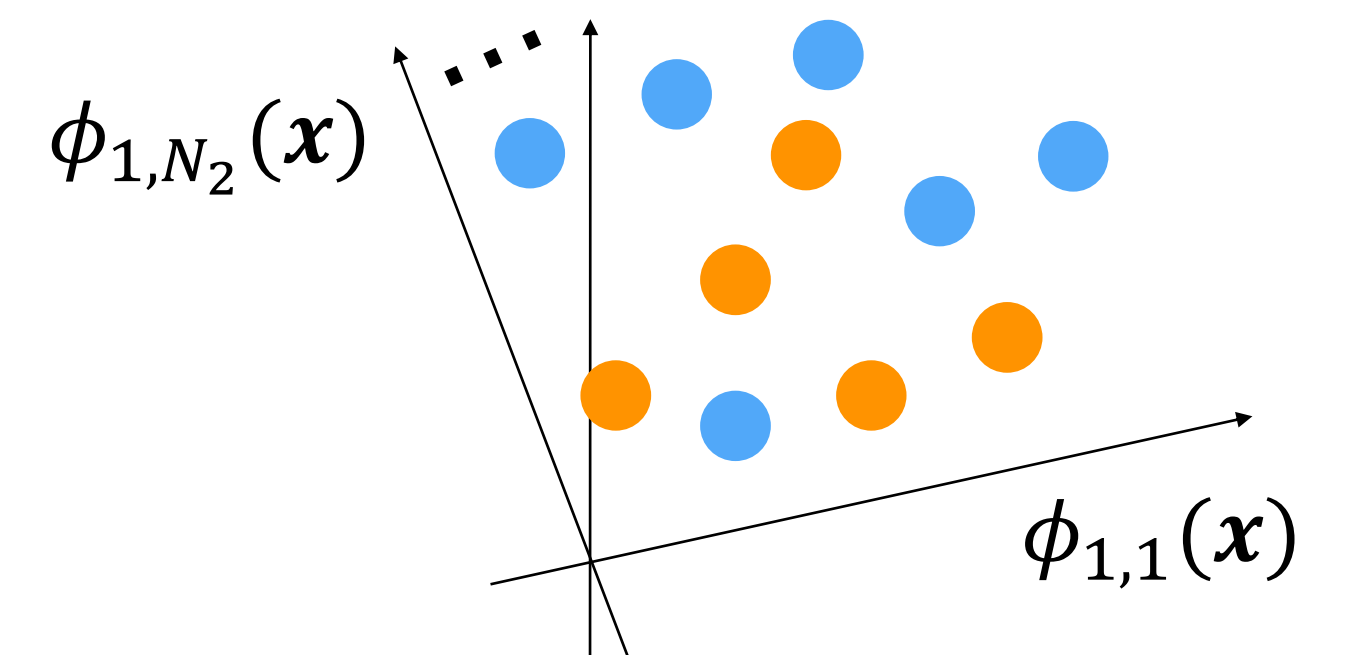
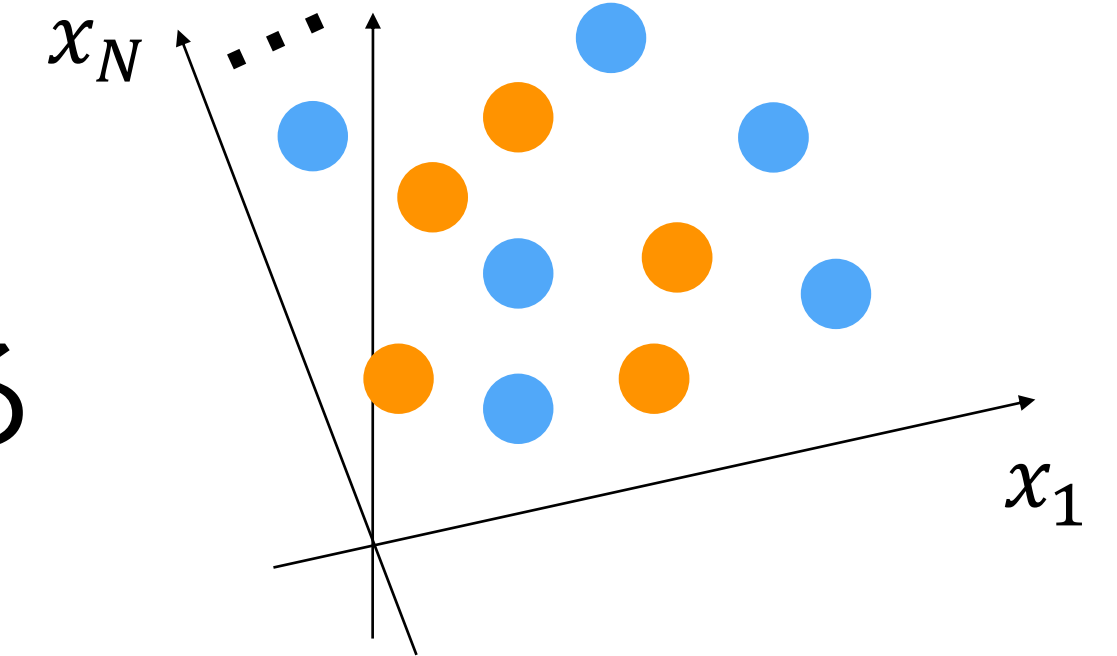
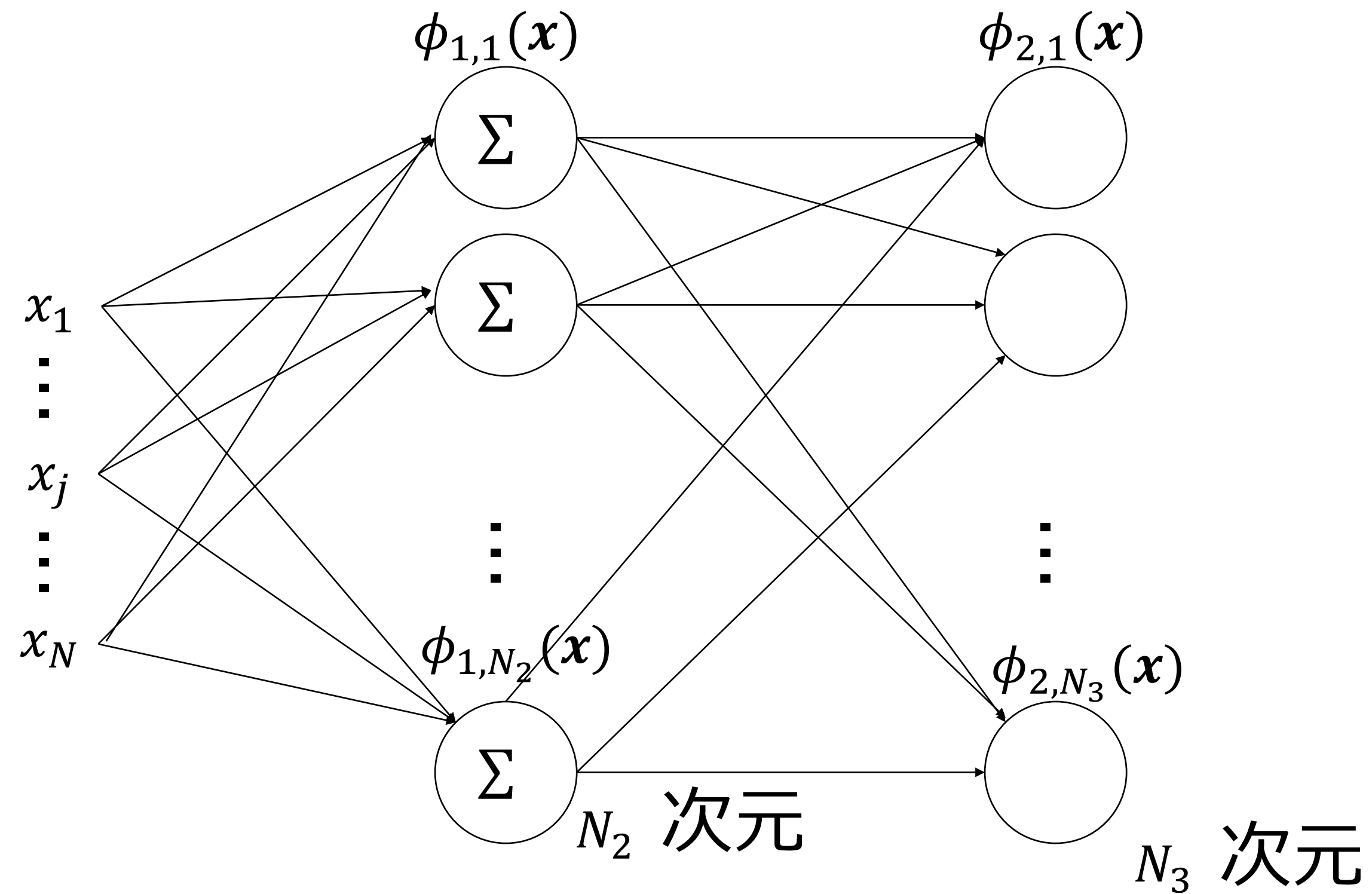
SVMなどと同じ
線形関数！（内積）

非線形関数



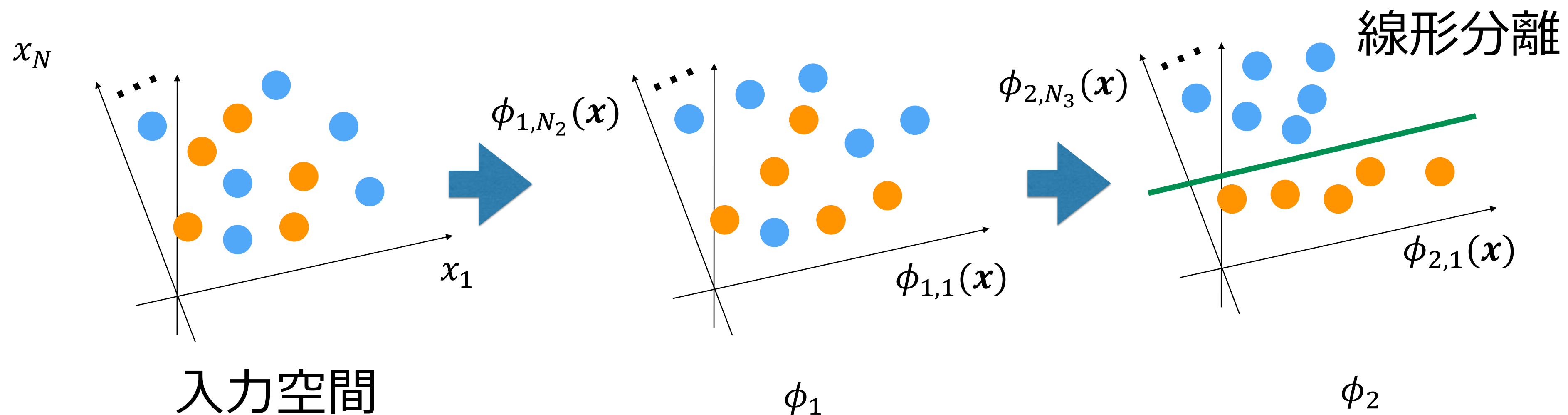
多層にすると...

- 出力を特徴写像空間としてさらに写像を重ねられる



つまるどころ

- 特徴写像を何度も重ねることで線形分離しやすい空間に写像
- 最終層だけ見ればSVMなどの分類器と同様
- 中間層で特徴も学習できるのがニューラルネットワーク



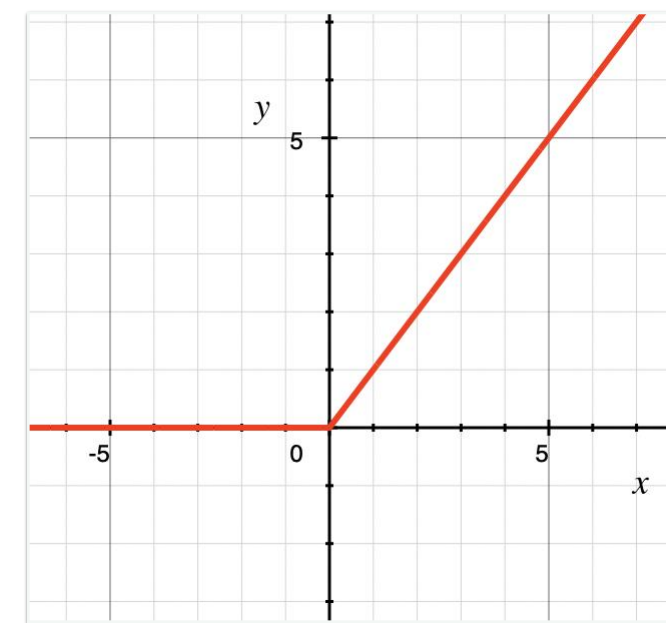
活性化関数

- さっきの議論だけでは不十分で、線形変換を何度繰り返してもできあがるのは線形変換
 - ◆ 各ノードで活性化関数による非線形変換を行う
 - ◆ 非線形変換を行っているのが「活性化関数」

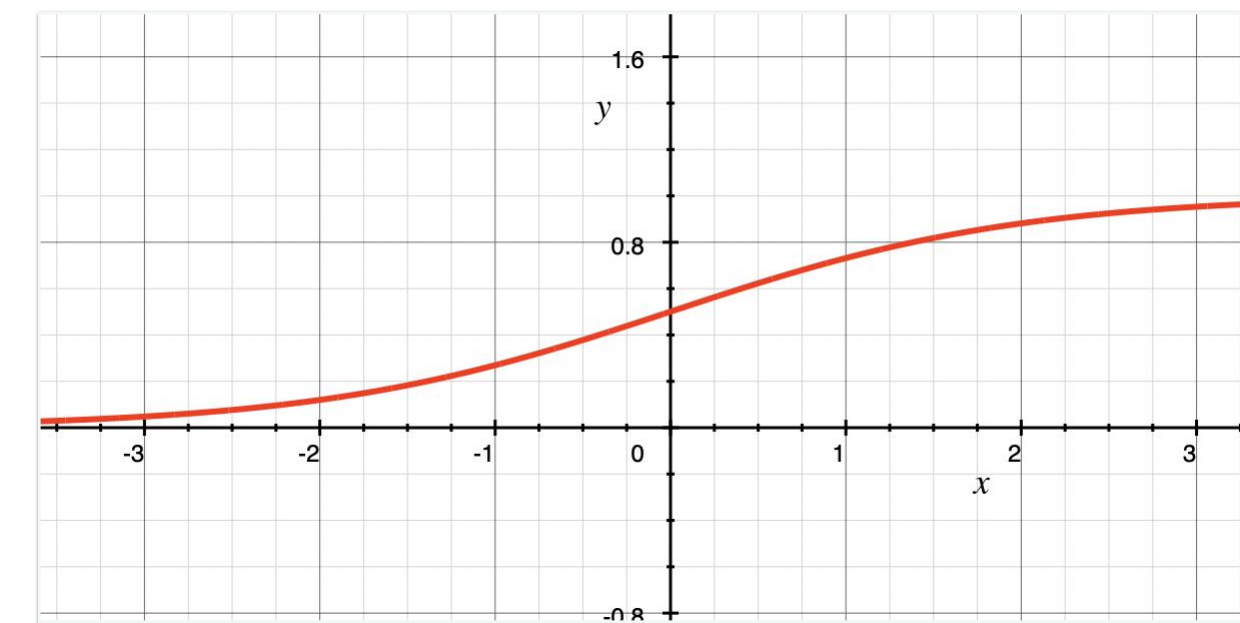
- よく用いられる活性化関数

- ◆ ReLU: $\sigma(x) = \max(x, 0)$

- ◆ シグモイド: $\sigma(x) = \frac{1}{1+e^{-x}}$



ReLU



シグモイド

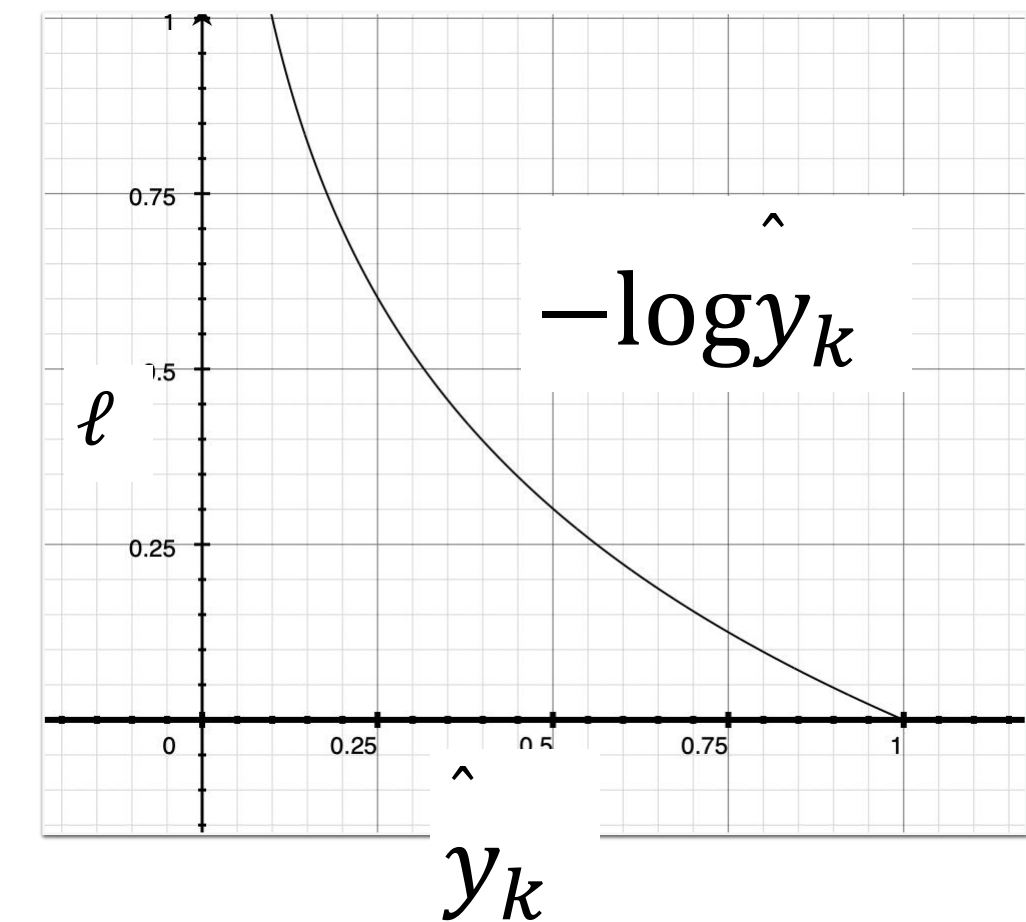
活性化関数は性能にどう影響する？

- ◆ 性能や最適化の安定性に影響がある
- ◆ このサーベイ
<https://arxiv.org/pdf/2402.09092>
によると、400個くらいあるらしい
- ◆ このサーベイ
<https://arxiv.org/pdf/2109.14545>
によると、性能面でも結構違いがあるらしい

決定版はないが、今のところ最も使われているのは ReLU
(および ReLUの拡張版)

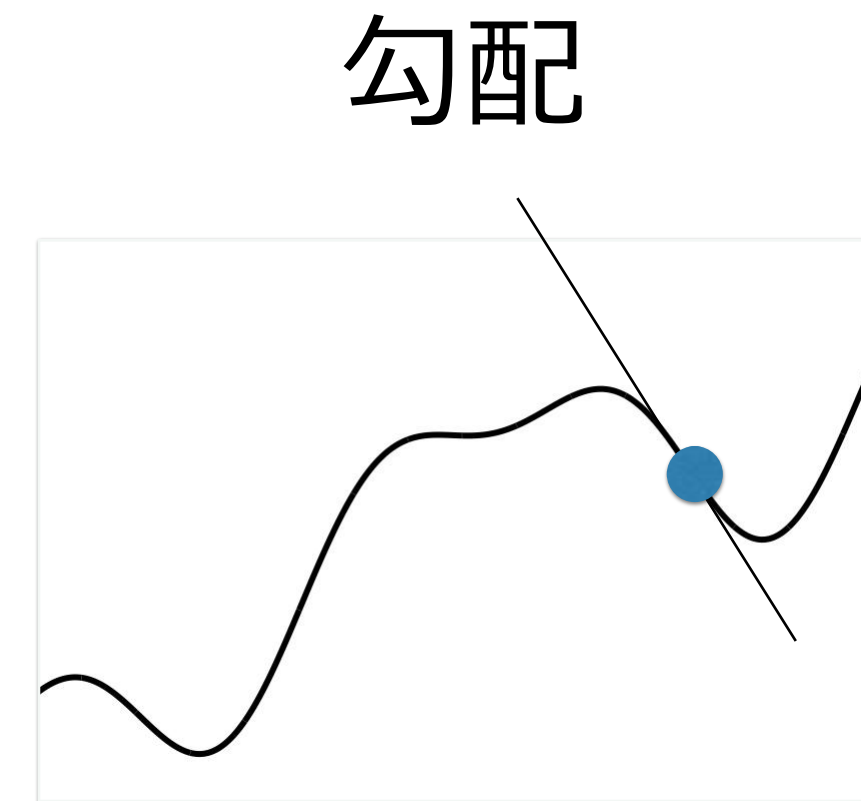
学習の仕方

- K クラス分類の場合
- ラベルは one-hot ベクトルに変換する
例： $\mathbf{y} = (0,0,1,0,0)$ だとするとラベルは3
- クロスエントロピー損失
 - ◆ $\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^K y_k \log \hat{y}_k$ (\hat{y}_k はモデルの出力. クラス k である確率)
 - ◆ $y_k = 1$ のとき, クラス k の確率 \hat{y}_k が1に近いほど損失が小さい
- 基本的にはこの損失の最小化 (場合によってはプラス様々な正則化項) を
勾配降下法で何度も繰り返す (繰り返し数をエポック数などと呼ぶ)
 - ◆ 正確には, 訓練サンプル「ひとなめ」が1epoch

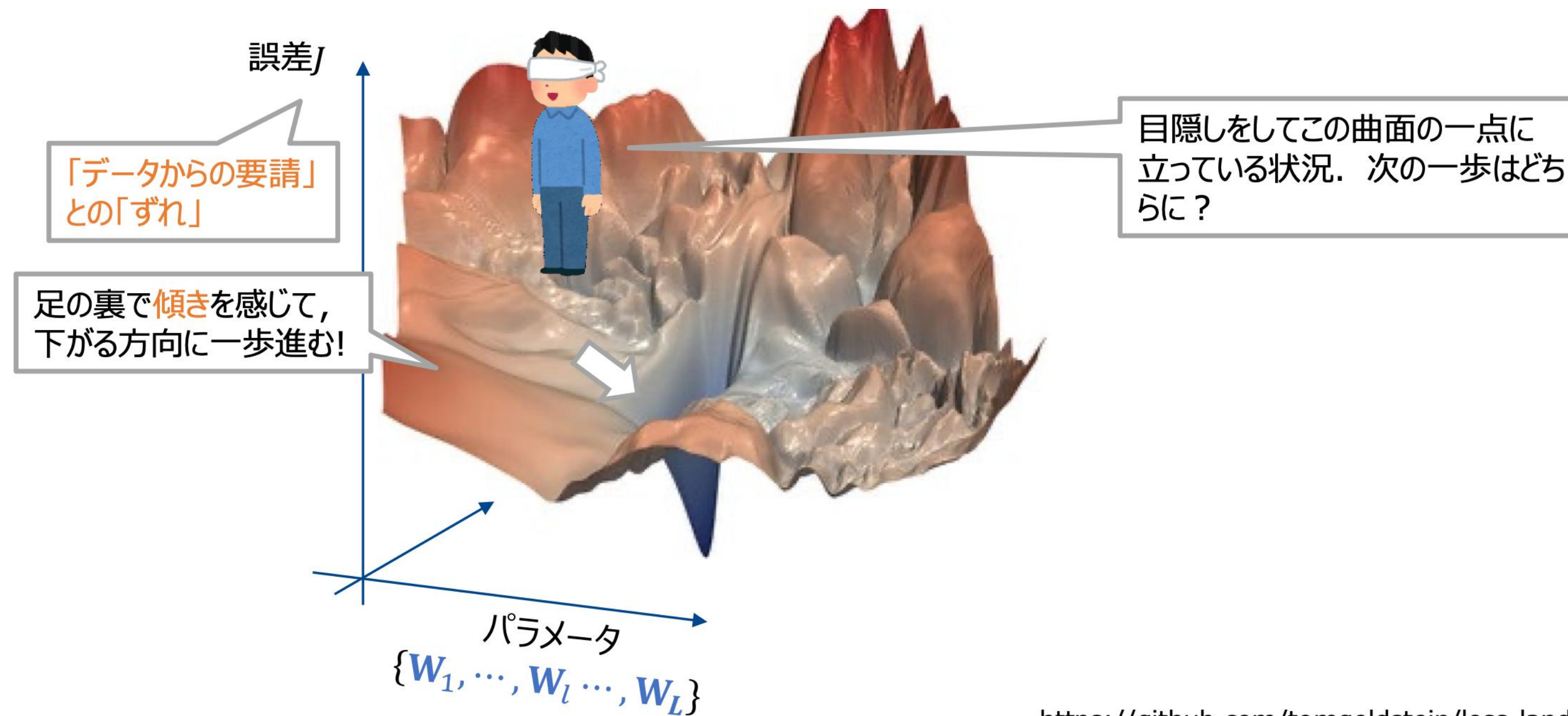


学習の仕方

- 当然ながら目的関数は「非凸」
- 基本的には勾配降下法で局所解を求める
 - ◆ SGD (確率的勾配降下法)
 - ハイパーパラメータの学習率を調整する必要あり
 - ◆ Adam [Kingma&Ba, 2014]
 - 最適化の安定化 & 学習率を自動調整する (良い解に到達しやすい)
 - 一方で他のハイパーパラメータが増える (みんな大体デフォルト値を使っているが調整すると改善することもある)
 - ◆ 実験的には Adam の方が性能が良いことが多いが、一概には言えない
 - ◆ 最近ではAdamWという改善手法が出ているのでAdamよりはそっちを使うほうが基本的に良い

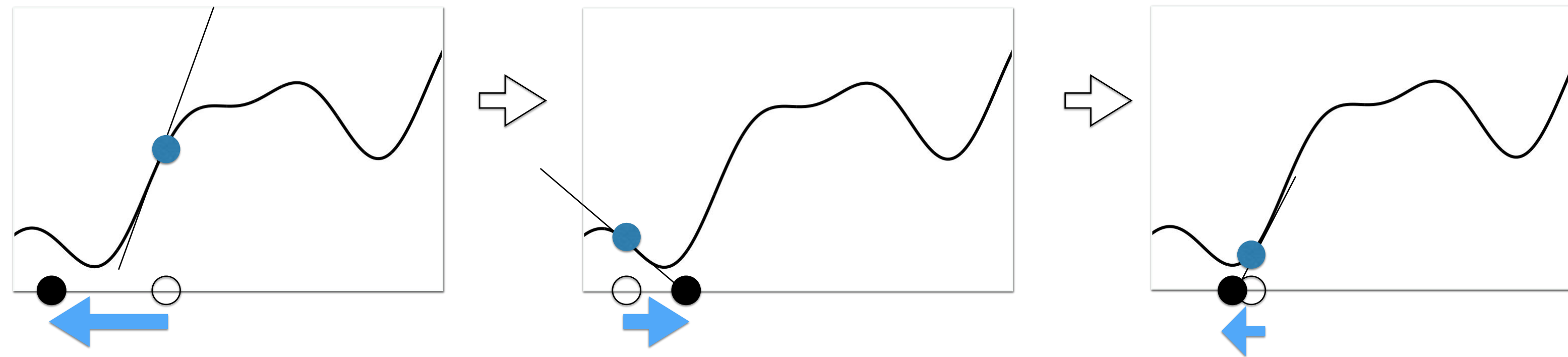


ニューラルネットワークの最適化のイメージ



<https://github.com/tomgoldstein/loss-landscape>
[Li+, "Visualizing the Loss Landscape of Neural Nets," NIPS2018]

勾配降下法

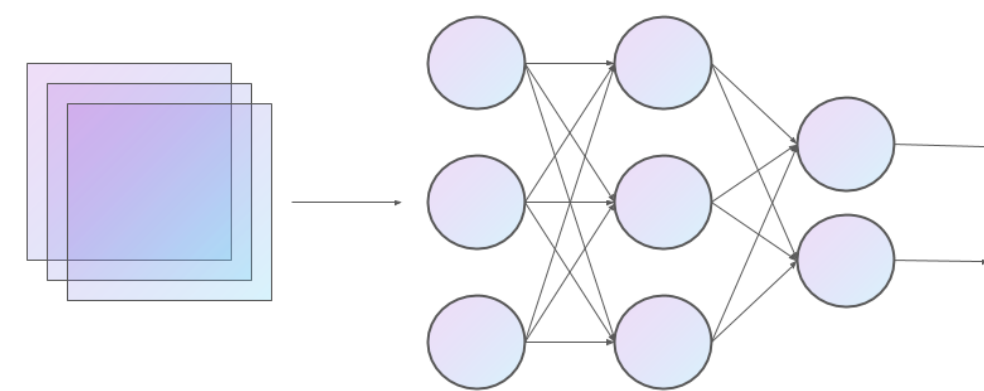
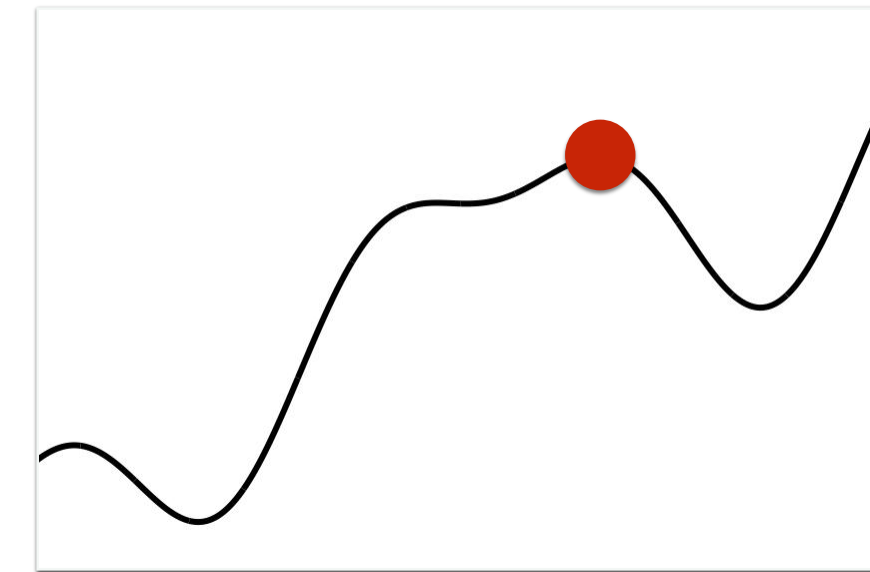


移動量
(ステップ幅, 学習率)

徐々に極小値に収束

難しさ

- 勾配が0になる鞍点の存在
 - ◆ 各山のてっぺんも勾配0
- 勾配の不安定性
 - ◆ 平坦だったり, 急斜面だったり
- 正則化, ドロップアウト (最適化の各ステップでノードをランダムに削除), 事前学習 (良い初期値の設定) などを行うことで回避



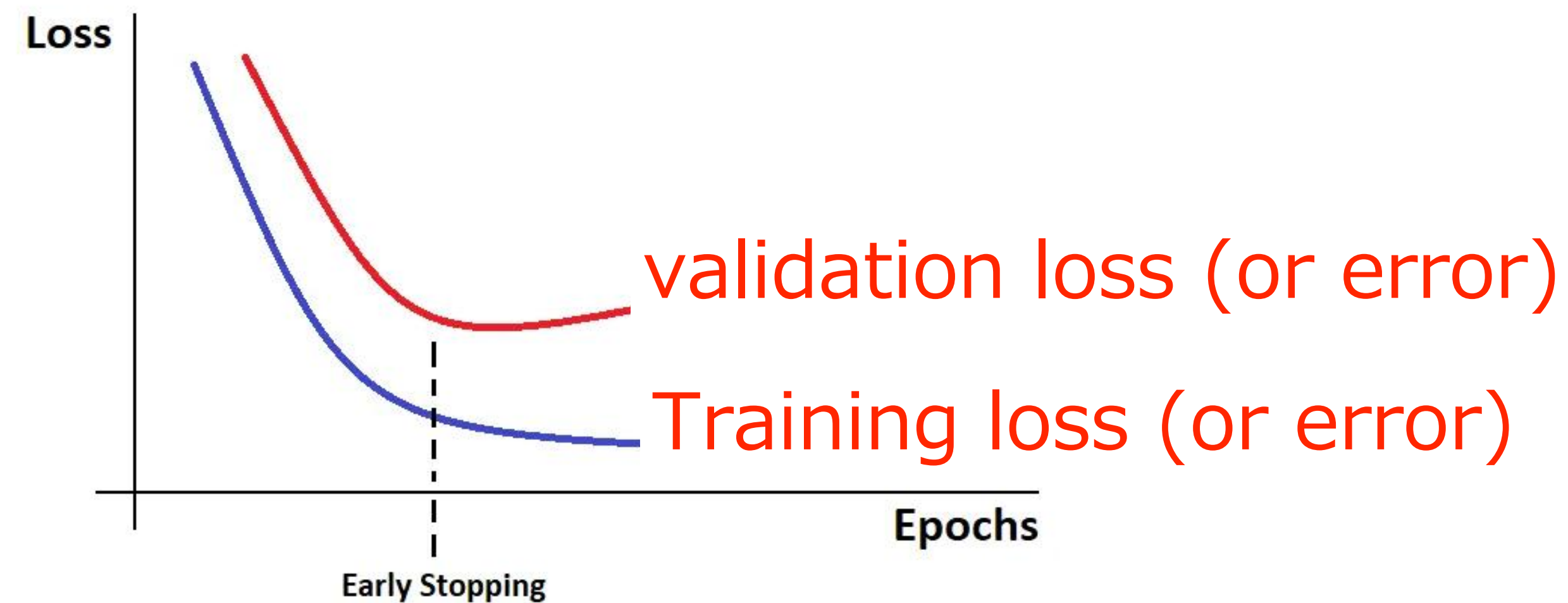
ドロップアウト：ランダムにノードを削除したり戻したりすることで特定のノードに依存しすぎないようにする

ニューラルネットの学習に向けて

- 目的（分類, ランキング, 回帰, etc.）に応じて損失関数を定義
- お好きなニューラルネットワークの構造（アーキテクチャ）を準備
 - ◆ 画像の場合は Convolutional Neural Network (CNN, 「畳み込み層」やプーリング層を用いたもの) を使うと良いと言われる（画像の局所的な特徴を抽出することができる）
 - ◆ ResNet などが有名
 - ◆ 言語だと transformer とされるものが有名
 - ◆ 他にも Auto-encoder（圧縮画像を出力, PCAのような役割）, U-Net（画像を入力して画像を出力. 領域抽出などに使われる）, などなど多様なDNNがあるので目的に応じて調べて使うと良い
- ハイパーパラメータを設定
 - ◆ 学習率, 正則化, 最大エポック数
 - ◆ 他にも細かいものがたくさんあるが割愛

Early stopping

- 最適化を繰り返すすぎると過学習することと言われる
- 適当なタイミングで終了する必要がある (early stopping)
- 最も単純なやり方は評価用データ (後述) を使った方法

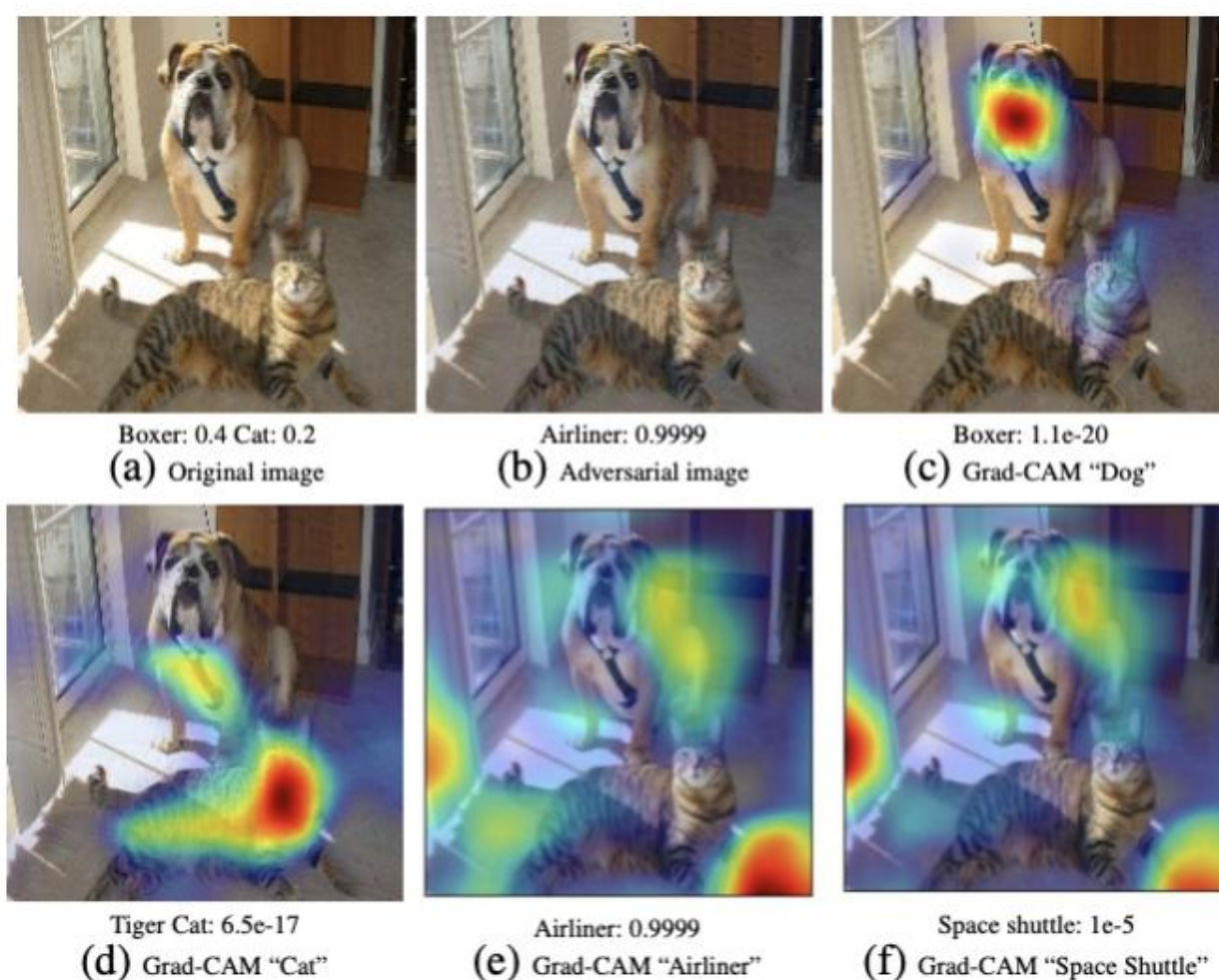


ここでのパラメータを採用

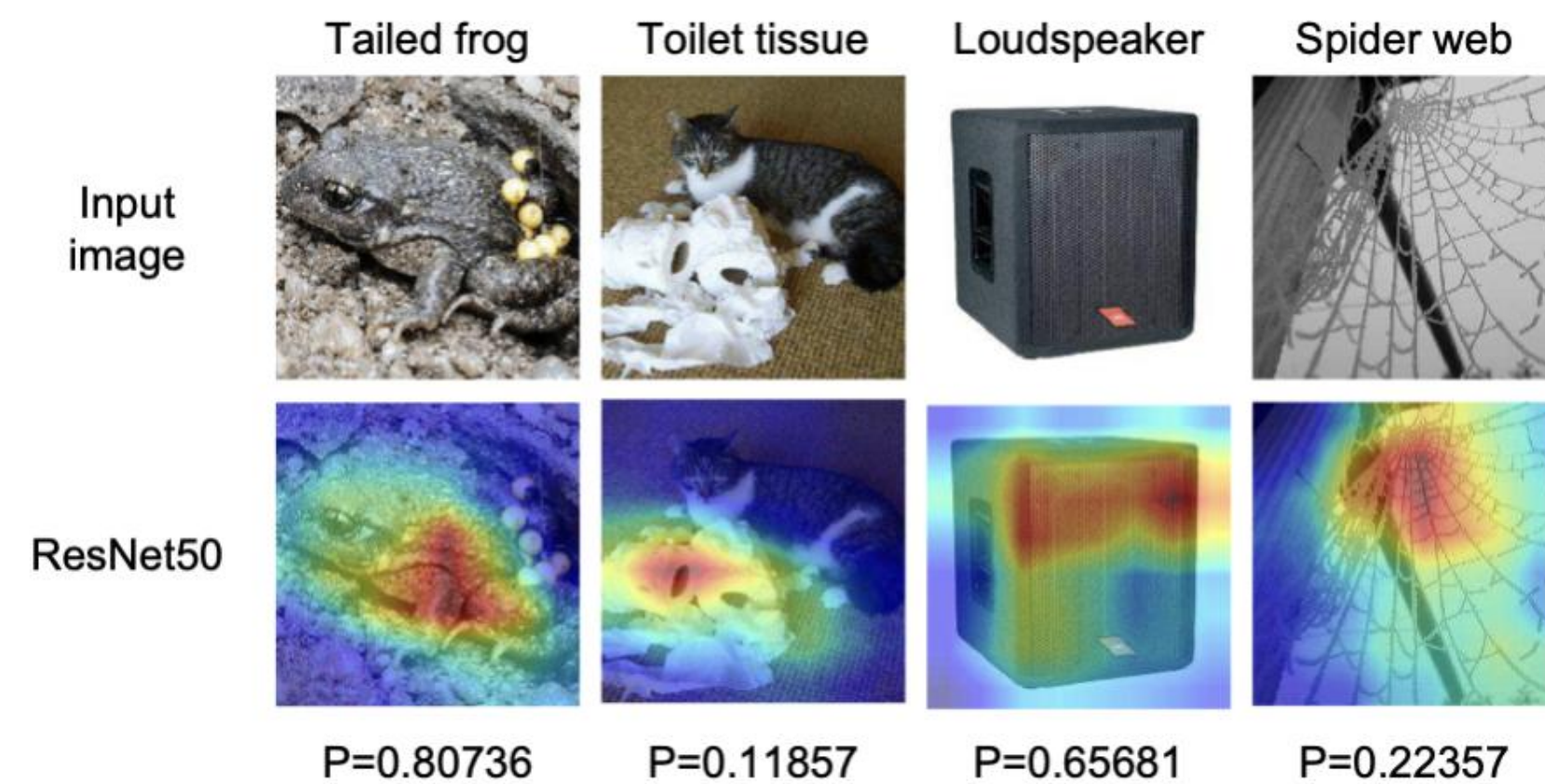
NNの解釈性

- 画像認識などの分野では解釈性を与える技術が進歩している
- Grad-CAM [Ramprasaath+, '17] などは有名
- まだまだ発展途上の部分も多いが、かなり研究が盛ん

Grad-CAM



attention map の例 [Woo+, '18]



Transformer

- 特徴間の類似度を網羅的に抽出可能
 - ◆ 自己注意機構（Self-Attention）が特徴的
- 従来の畳み込みNN（CNN）などは局所的な特徴が得意だが、離れているとダメ
- 「今日のランチは豚骨ラーメンだった。店は混んでたなあ。でも美味しかった。」
 - ◆ → 「豚骨ラーメン」と「美味しかった」が離れているので、CNNのような局所的な特徴をつかむ学習だと関連性を掴めない
 - ◆ Self-attention では全ての単語間の類似度を使って学習するので、局所はもちろん大域的な特徴もつかめる
- 画像などでもうまくいく例が多々（Vision Transformer）
 - ◆ さらには入力データの種類が複数あっても関連性を学習可能

「マルチモーダル」の主流化

- 従来はシングルモーダル（入力データの種類が1つ）が主流
- 現在はマルチモーダル（例：画像，音声，テキスト）から予測等を行うことが可能に
 - ◆ 基本的には Transformer ベース
 - CLIP (Contrastive Language-Image Pretraining) :
テキストと画像で学習し特徴空間を統合
 - Flamingo : 画像と言語
 - ◆ ただし最近ものすごいスピードでモデルがアップデートされているので，明日には古い方法となっている可能性も...

導入の注意点

- Transformer はニューラルネットワークと同様（かそれ以上に）計算コストが大きい
 - ◆ 通常数百GBのGPUメモリや高性能なGPUが必須
 - ◆ 電気代もスゴイ...
- お試し利用→クラウドサービスを利用
 - ◆ AWS (Amazon) , ABCI (産総研) , 大学のHPCなど
- 最先端の方法に違いはないですが、目的や実化環境、自分のデータとも相談しながら適切な機械学習手法を探しましょう

ニューラルネットワークのメリット・デメリット

■ メリット：

- ◆ 複雑な特徴を必要とするタスクには非常に有用
 - 画像や言語などではほどニューラルネットワークの独壇場
- ◆ ライブラリの整備
 - Python に大きな偏りはあるが、便利な関数や方法のコードが整備されている
 - 最新方法もgithubなどで公開されていることが多い（ただし動作責任は自身で）

■ デメリット：

- ◆ 過学習を回避する理論・技術が確立されていない
- ◆ ハイパーパラメータのチューニングが大変
 - ハイパーパラメータのチューニングアルゴリズムなども色々提案されているので活用するのもあり
- ◆ 解釈性が弱い（画像などの一部では方法が提案されているが確立されていない）
- ◆ 大規模データではGPUが必須
 - 計算量と電気代、GPU代は大きいので最初は安価なクラウドサービスなどで始める

各学習器の比較まとめ

あくまでも目安です

	データの種類		データ数	学習コスト	推測コスト	複雑なデータ への対応力	解釈性
	連続	離散					
近傍法	○	△	膨大に必要	なし	大	×	○
SVM	○	△	少量で良い (数百~数千)	線形の場合は小	小	△	○ (線形の場合)
Boosting, Random Forst	○	○	少量で良い (数百~数千)	小	小	△	○
DNN	○	△	膨大に必要 (数万~)	膨大 (GPUが必要)	GPU必要	○	×

シチュエーション例

- 不良品の改善：特徴量や説明変数が大量にある（高次元データ）ので、有効な説明変数を絞り出して原因を突き止め改善したい
 - ◆ 改善が最終目的で解釈性が必須のため、まずはSVMが良い
- 医療や教育での仮説ベースタスク：自分で仮説や説明変数を用意していて、それらが分類などに有効か機械学習で確認したい
 - ◆ 解釈性が必須なのでSVMやBoosting, RFが良い
- 分析初期：今のところデータ数があまりないけど、性能が出そうならデータをさらに採っていききたい
 - ◆ まずはSVMやBoosting, RFで始めるのが良い
 - ◆ データ数が集まればDNNもあり
- データが離散値ばかり
 - ◆ BoostingやRFなどの決定木ベースの手法が良い
- 性能命：とりあえず性能が欲しい
 - ◆ まずはSVM, Boosting, RFで始めて、厳しそうならばDNNにシフト

演習

ニューラルネットワークの実践

- GPUの使用が推奨
 - ◆ google colab であれば ランタイム→ランタイムのタイプを変更→GPU or TPU
 - ◆ 小さめのニューラルネットワークであればCPUでも大丈夫ですが、大きくなると計算時間が膨大になる
- プラットフォームは色々ある
 - ◆ keras (古めだがベーシック) , Tensorflow (Google) , Pytorch (メタ) 等
 - ◆ 最近は Pytorch を使う人が多い印象
- ニューラルネットワークは計算時間が大きいので、交差検定は省略して固定の評価用データでモデル選択をすることが多い
 - ◆ ただし汎化誤差を正當に評価するため、テストデータは交差検定等を通して実験しよう
- 今回はMNISTデータを利用して実験

アーキテクチャごとの性能比較

- 小さなMLP (多層パーセプトロン)
- 中くらいのMLP
- CNN (畳み込みニューラルネットワーク)
 - ◆ 画像の局所特徴を重視

```
Small MLP: Test Acc = 0.9746, Training Time = 35.60 sec  
Medium MLP: Test Acc = 0.9752, Training Time = 44.77 sec  
CNN: Test Acc = 0.9854, Training Time = 46.90 sec
```

画像データなので、CNNが有効に働いている

ハイパーパラメータ等の影響

- 学習率
 - ◆ 勾配方向にどれくらい進むか
 - ◆ 適切な値にしないと良い解に収束しない
- ドロップアウト
 - ◆ 学習中（最適化中）ランダムにノードを削除することで過学習を回避する方法

学習率：同じ10epochでも...

0.01 accuracy: 0.9777 - loss: 0.0777 - val_accuracy: 0.9723 - val_loss: 0.1373

0.001 accuracy: 0.9945 - loss: 0.0174 - val_accuracy: 0.9747 - val_loss: 0.1008

0.0001 accuracy: 0.9721 - loss: 0.1003 - val_accuracy: 0.9660 - val_loss: 0.1115

データの規模やアーキテクチャによっては
1epochの学習時間が膨大
→ 学習率は計算コストの面でも非常に重要

ドロップアウト

なし `accuracy: 0.9946 - loss: 0.0177 - val_accuracy: 0.9804 - val_loss: 0.0829`

あり `accuracy: 0.9793 - loss: 0.0642 - val_accuracy: 0.9789 - val_loss: 0.0734`

ドロップアウトをしても残念ながら評価精度は向上せず
(そんなこともあります)

しかし、訓練精度と評価精度のギャップはドロップアウトをしたほうが小さい
(過学習を回避できている)

演習：学習器の選び方

まずは簡単な比較実験から

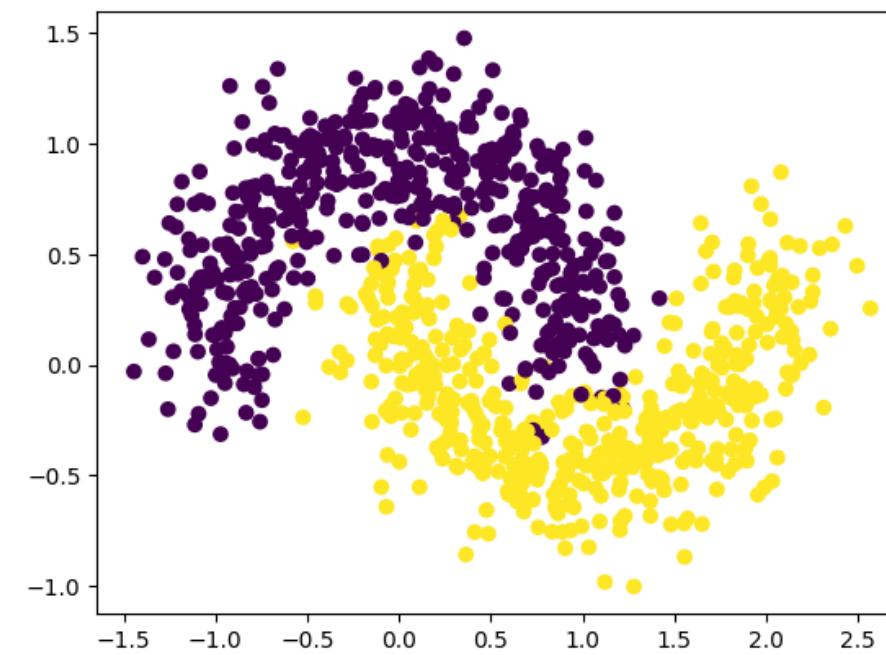
- 学習器で本当に性能差が出るのか？
- 色々な学習器を比較
 - ◆ SVM (非線形データに対しては kernel='rbf' として非線形学習)
 - ◆ Random Forest
 - ◆ AdaBoost
 - ◆ XGBoost
 - ◆ Neural Network
- 色々なデータで比較
 - ◆ 人工データ (線形 (データ数5000) , 非線形 (1000))
 - ◆ 表データ (wineデータ)
 - ◆ 画像データ (MNIST, CIFAR-10)

人工データ（線形）の結果

	Model	Train Accuracy	Test Accuracy	Train Time (s)
0	SVM	0.90075	0.860	8.315526
1	RandomForest	0.96725	0.914	8.308892
2	AdaBoost	0.88925	0.887	11.456553
3	XGBoost	0.94700	0.916	3.489185
4	DNN	0.95675	0.864	12.427943

表形式のデータではXGBoostはかなり強い

人工データ（非線形）の結果



	Model	Train Accuracy	Test Accuracy	Train Time (s)
0	SVM	0.96875	0.980	0.146588
1	RandomForest	0.99875	0.975	1.505263
2	AdaBoost	0.97750	0.965	1.698795
3	XGBoost	0.99000	0.975	0.569227
4	DNN	0.96625	0.980	6.438161

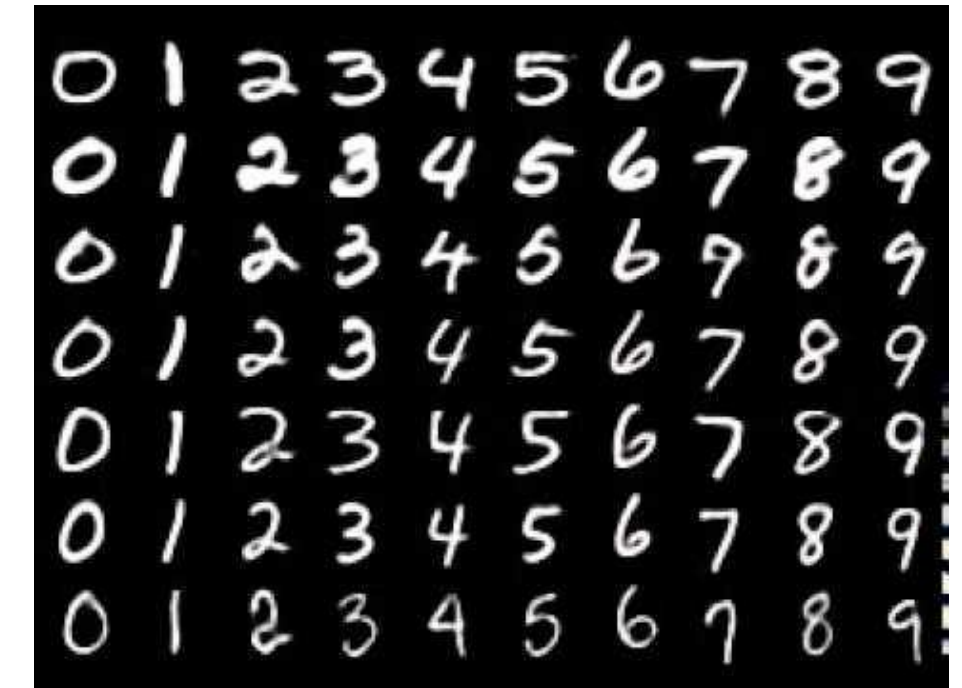
表形式かつ非線形データではカーネルSVMが有効になることも

wine データ

	Model	Train Accuracy	Test Accuracy	Train Time (s)
0	SVM	0.964789	1.000000	2.867915
1	RandomForest	1.000000	1.000000	2.067364
2	AdaBoost	1.000000	0.944444	1.577614
3	XGBoost	1.000000	0.972222	0.813879
4	DNN	0.647887	0.750000	5.201874

DNNは今回の表形式のデータにはイマイチ
(※表データに対するDNNも色々研究はされています)

MNISTデータ



- 都合上5000個にデータを減らしています
(オリジナルは50000個)

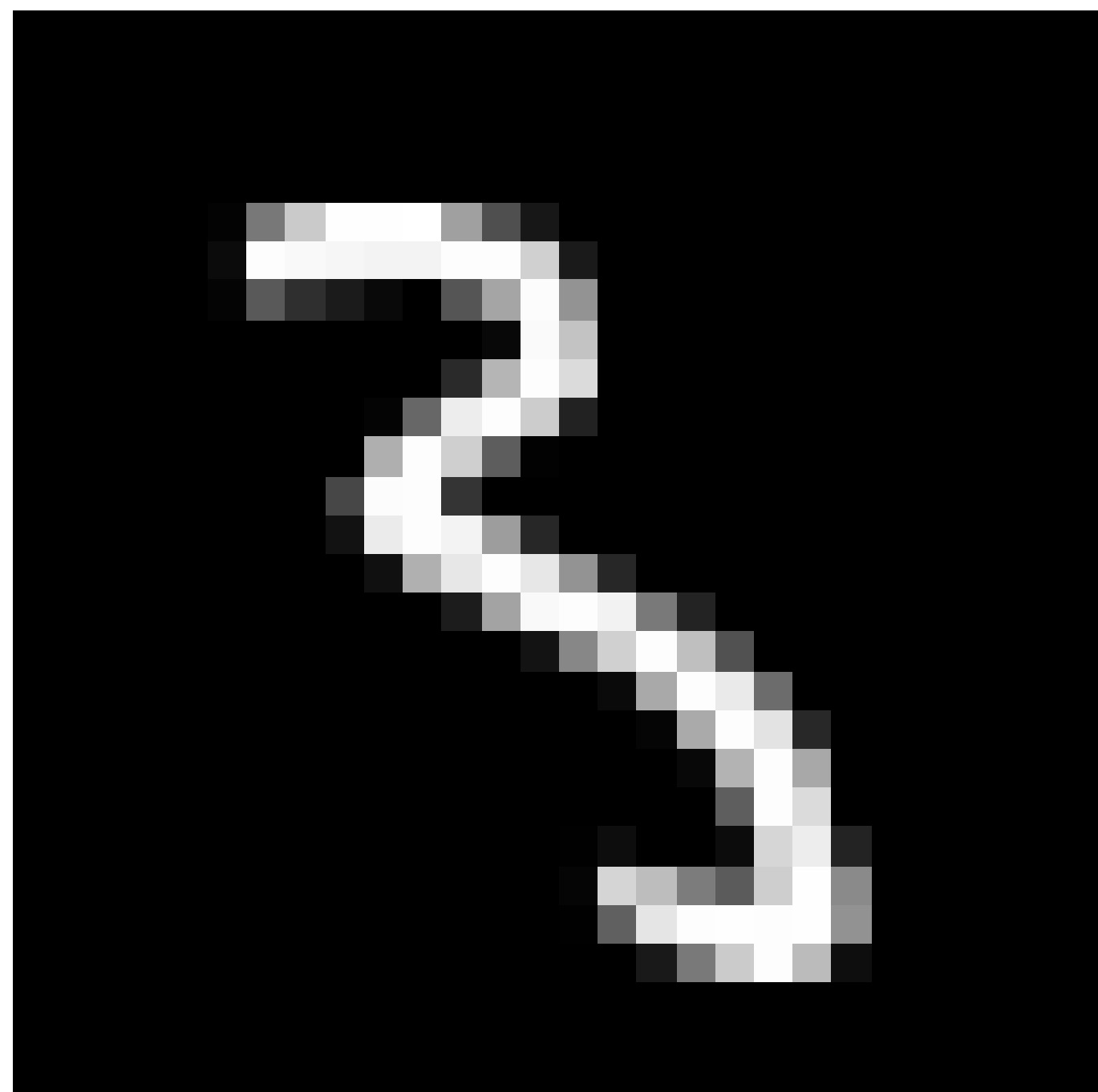
	Model	Train Accuracy	Test Accuracy	Train Time (s)
0	SVM	1.00000	0.956	39.768785
1	RandomForest	0.99550	0.931	11.024098
2	AdaBoost	0.77375	0.759	46.710219
3	XGBoost	1.00000	0.930	462.229679
4	DNN	1.00000	0.948	13.526012

XGBoostは特徴数が多いと決定木の学習に膨大な時間を要する

誤分類データの確認

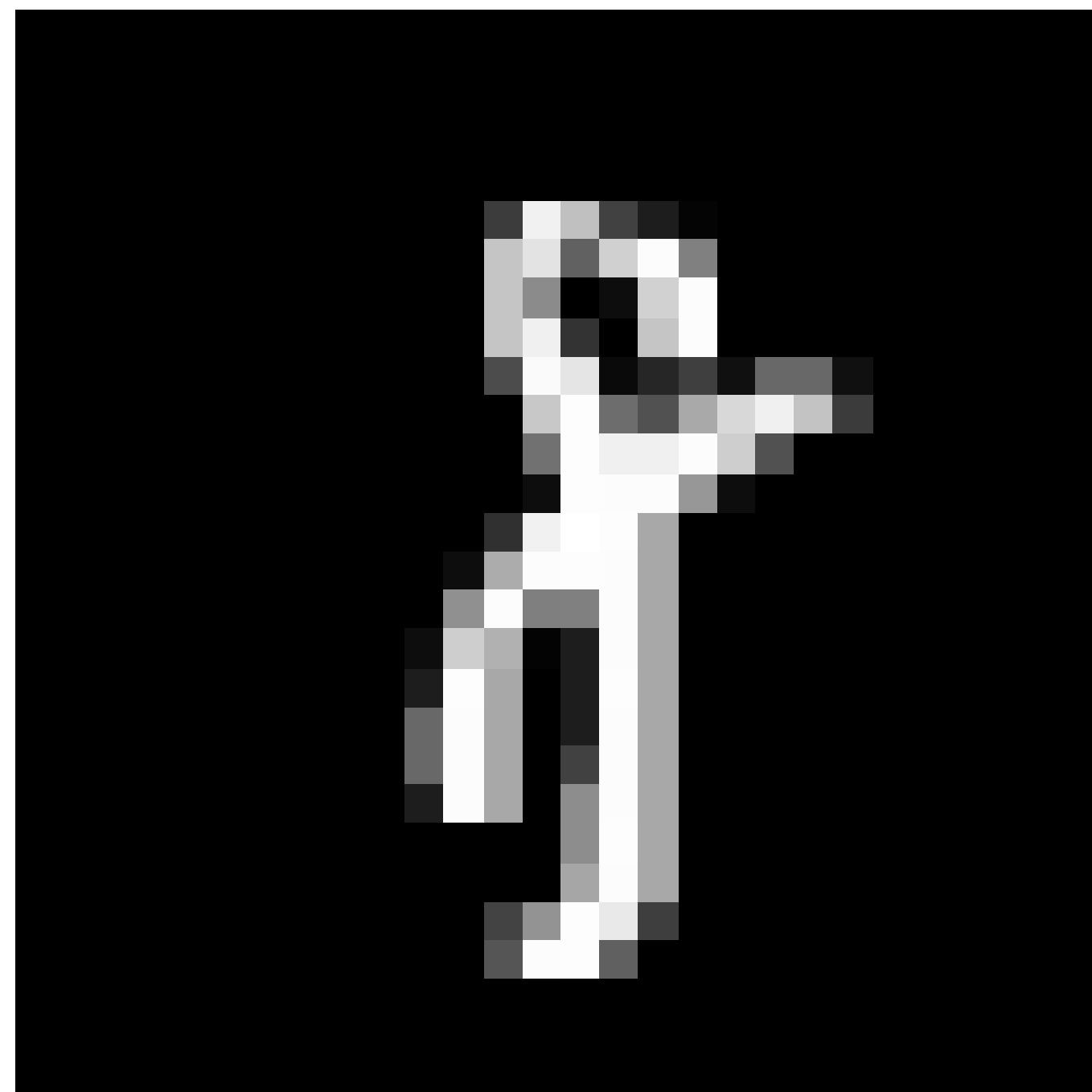
- どのようなデータを間違っているのか見てみる

True: 3, Pred: 5



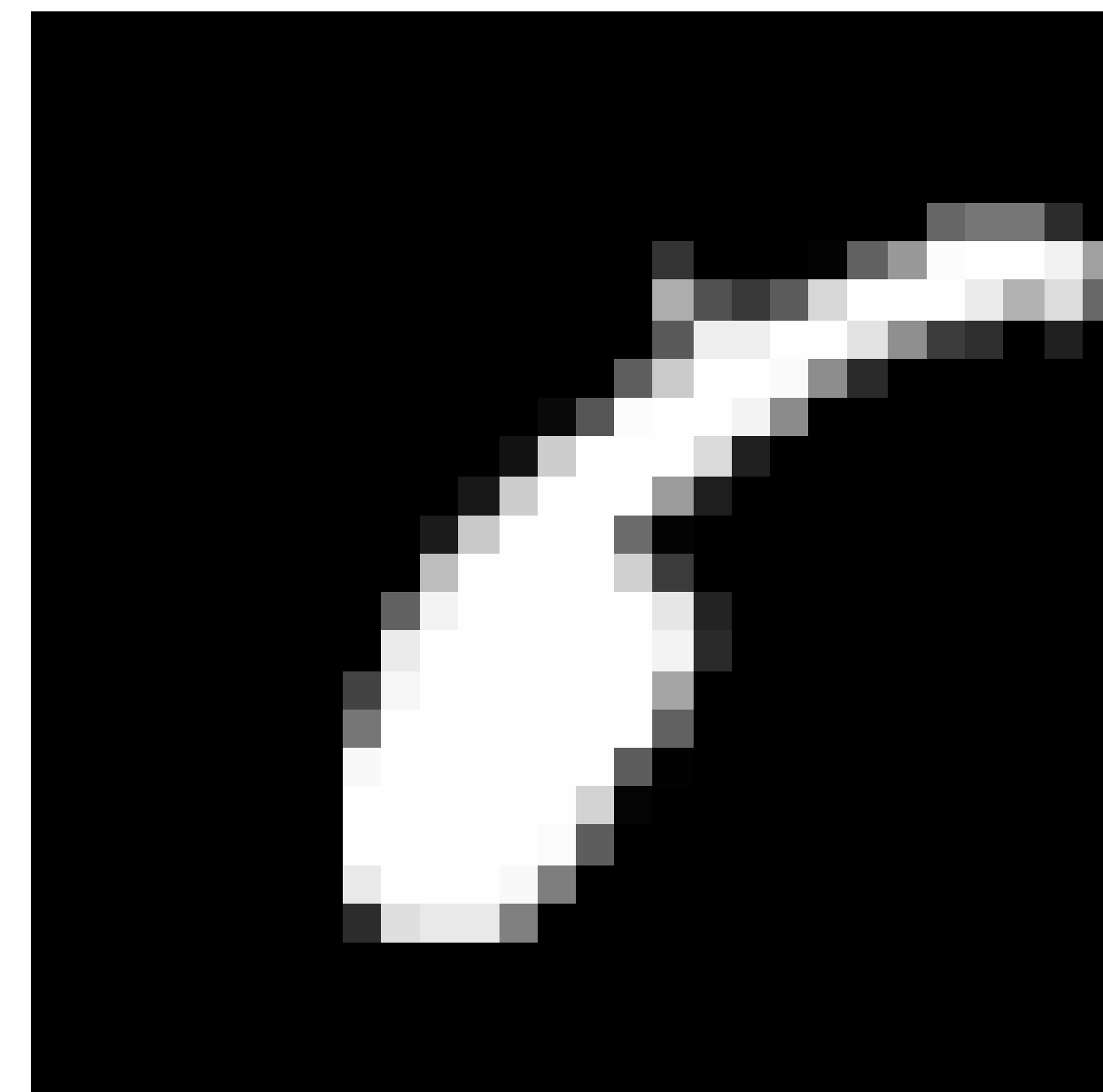
SVMの誤り例

True: 8, Pred: 1



RFの誤り例

True: 6, Pred: 5

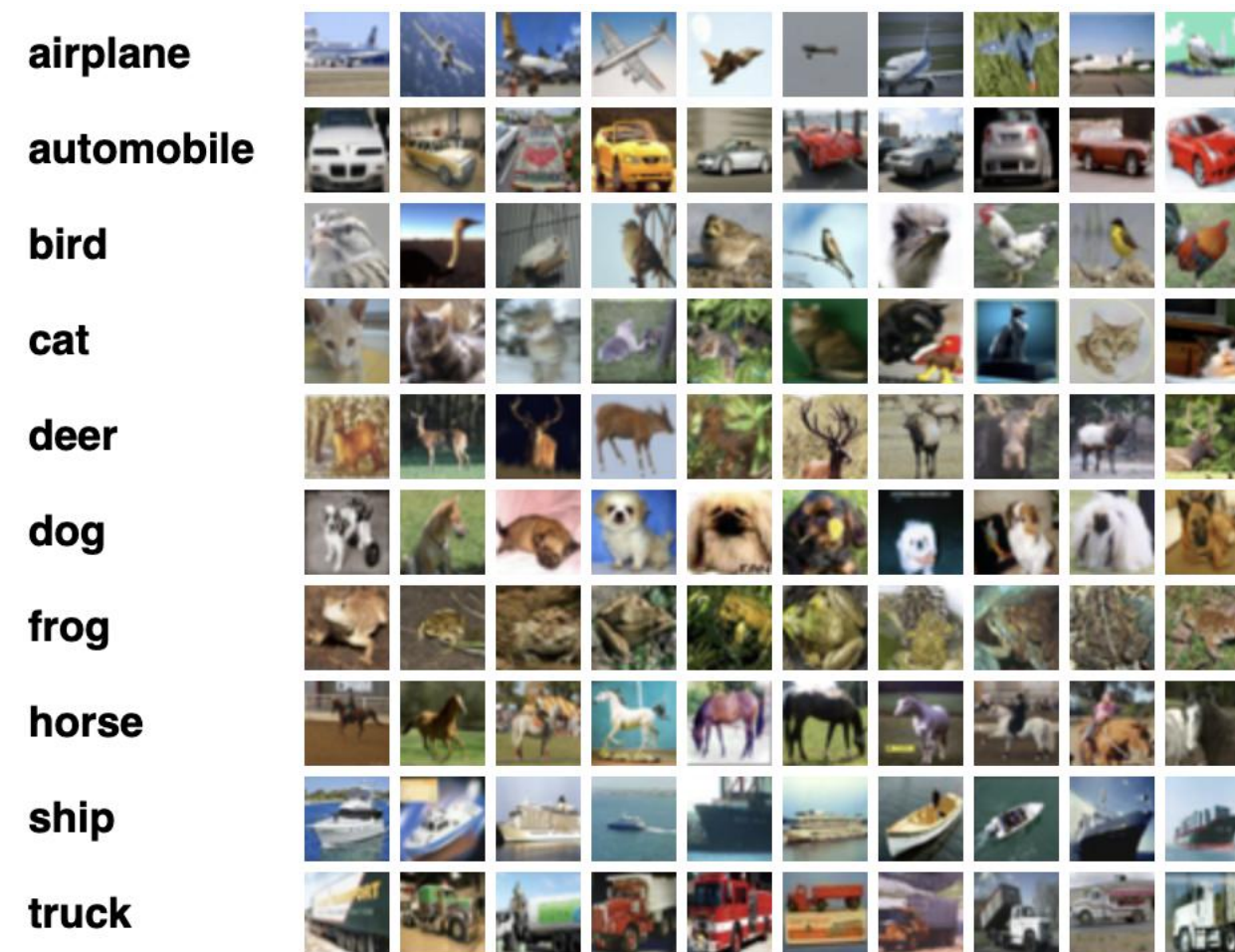


DNNの誤り例

CIFAR10

XGBoostはかなり時間がかかるのでコメントアウト推奨

- 都合上10000個にデータを減らしています（オリジナルは50000個）



<https://www.cs.toronto.edu/~kriz/cifar.html>

	Model	Train Accuracy	Test Accuracy	Train Time (s)
0	SVM	0.994375	0.4770	1384.250490
1	RandomForest	0.909625	0.3845	99.261878
2	AdaBoost	0.289625	0.2725	912.242178
3	XGBoost	1.000000	0.4725	6427.040976
4	DNN	0.287500	0.2655	27.166416

※手違いでAdamで最適化した結果です

あれ, DNNが低い...

CNNだと性能アップ

注：「ランタイム」でGPUに変更してやっています

```
Num GPUs Available: 1
✓ GPU is available and will be used for training!
Training CNN on GPU (Detected 10 classes)...
Epoch 1: Train Loss = 4.2521, Train Acc = 0.1235, Val Loss = 2.2334, Val Acc = 0.1580
Epoch 2: Train Loss = 2.2019, Train Acc = 0.1600, Val Loss = 2.0353, Val Acc = 0.2630
Epoch 3: Train Loss = 2.0680, Train Acc = 0.2116, Val Loss = 1.8804, Val Acc = 0.2830
Epoch 4: Train Loss = 1.9193, Train Acc = 0.2821, Val Loss = 1.6901, Val Acc = 0.3700
Epoch 5: Train Loss = 1.7867, Train Acc = 0.3315, Val Loss = 1.6388, Val Acc = 0.4070
Epoch 6: Train Loss = 1.6900, Train Acc = 0.3671, Val Loss = 1.5169, Val Acc = 0.4600
Epoch 7: Train Loss = 1.5825, Train Acc = 0.4156, Val Loss = 1.5720, Val Acc = 0.4345
Epoch 8: Train Loss = 1.4834, Train Acc = 0.4555, Val Loss = 1.4630, Val Acc = 0.4805
Epoch 9: Train Loss = 1.4141, Train Acc = 0.4779, Val Loss = 1.3794, Val Acc = 0.4950
Epoch 10: Train Loss = 1.3135, Train Acc = 0.5188, Val Loss = 1.3385, Val Acc = 0.5215
Epoch 11: Train Loss = 1.2620, Train Acc = 0.5345, Val Loss = 1.3449, Val Acc = 0.5090
Epoch 12: Train Loss = 1.1954, Train Acc = 0.5641, Val Loss = 1.2900, Val Acc = 0.5330
Epoch 13: Train Loss = 1.1158, Train Acc = 0.5979, Val Loss = 1.2929, Val Acc = 0.5425
Epoch 14: Train Loss = 1.0554, Train Acc = 0.6096, Val Loss = 1.2903, Val Acc = 0.5590
Epoch 15: Train Loss = 0.9670, Train Acc = 0.6456, Val Loss = 1.3588, Val Acc = 0.5530
Epoch 16: Train Loss = 0.9411, Train Acc = 0.6461, Val Loss = 1.3639, Val Acc = 0.5595
Epoch 17: Train Loss = 0.9006, Train Acc = 0.6733, Val Loss = 1.3415, Val Acc = 0.5625
CNN: Final Train Acc: 0.6733, Final Test Acc: 0.5625, Time: 17.80s
```

より複雑なネットワーク（ResNetなど）を採用し
すべてのデータ（50000個）を使うと
もっと性能が上がります

※手違いでAdamで最適化した結果です

さいごに

どんなデータを使って、どんなことがしたい？

- 何を予測して何がしたい？（一番大事）
 - ◆ 分類：（データからカテゴリを予測する関数を求める）：医療画像からの疾病予測
 - ◆ 回帰（データから結果（多くは数値）を出力する関数を求める）：売上予測など
 - ◆ ランキング（順番付ができるような関数を求める）：商品推薦など
 - ◆ 異常検知：アブノーマルなデータを検知する：故障検知など
- データに対して「どんな答えが」or「答えがどの程度」与えられているか？
 - ◆ 教師あり学習（訓練データが全部回答付き）
 - ◆ 半教師あり学習（訓練データの一部が回答付き）
 - ◆ 弱教師あり学習（訓練データに回答のヒントが付いている）
 - ◆ 教師なし学習（訓練データに全く回答が付いていない）
- 訓練データの与えられ方
 - ◆ まとめて与えられる（オフライン）
 - ◆ 逐次的（オンライン）：日々予測しては結果を受け取る、株価予測など

すべてを細かく紹介するのは
難しいですが、機械学習は
「色々な状況で」「色々なことができる」
ので状況や目的に応じたアプローチをとるのが
重要です

機械学習でやる前に

- データをちゃんと「眺める」．必ず前処理をする
 - ◆ 異常値や欠損がないか
- 目的やデータ条件を「ちゃんと」定める
 - ◆ 目的：分類性能を上げたいのか，回帰がしたいのか，ランキング性能を上げたいのか，解釈して改善したいのか， etc.
 - ◆ データ条件：答えはすべてのデータについているのか，一部にしかついていないのか， など
 - ◆ 例：成績優秀者とそれ以外を分類できたとして，成績を上げるにはどうすればいいかが必要な場合は，学習器に解釈性が必須
- まずは簡単な問題からスタートしてみる
 - ◆ 例：9種類の病人と健康な人のデータがあったとき，いきなり10クラス分類ではなく健康な人と病人の2クラス分類から始めてみる
 - 2クラス分類ができないならそもそも厳しい（学習器を変える or 説明変数増やす or データ増やす）

大事なこと

- まずは目的設定が最重要
 - ◆ 学習して「良くしたい」ものは何？（分類誤差, AUC, precision, recall, etc.）
- やみくもに学習器を試すのではなく論理的思考が大事
- 性能が出ないのであれば、原因は何か？
 - ◆ 訓練データでも精度が出ない→特徴（説明変数）が足りない（色々な説明変数を収集する）、仮説クラスがシンプルすぎる（NNなど複雑な仮説を試す）
 - ◆ 訓練データでは性能が高いが評価用データに対する性能やテスト性能は低い→データ数が足りない（データ数を増やす、よりシンプルな仮説を選ぶ）、過学習（画像や言語など、NNが必須の場合は正則化やドロップアウトなどを試す）
 - ◆ （一番大事かも）バグを疑う