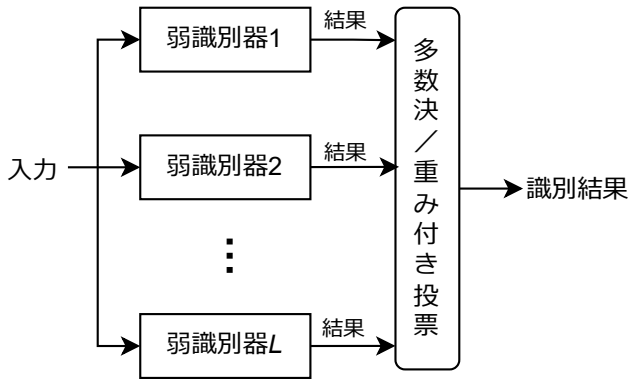
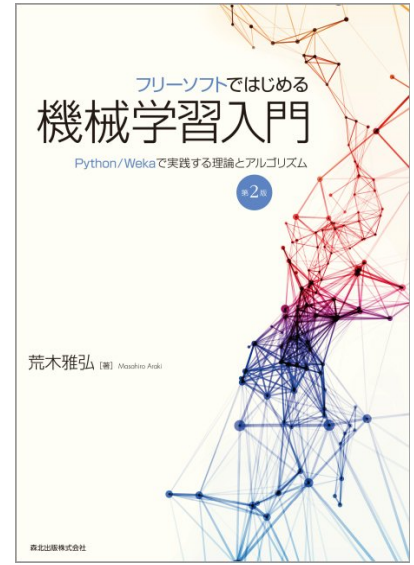
# 10. アンサンブル学習




- 10.1 なぜ性能が向上するのか
- 10.2 バギング
- 10.3 ランダムフォレスト
- 10.4 ブースティング
- 10.5 勾配ブースティング



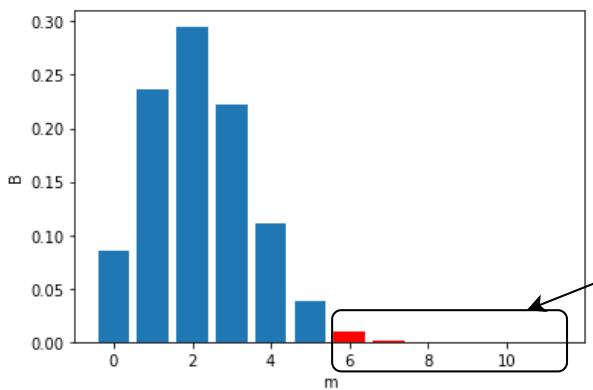

- 荒木雅弘:『フリーソフトではじめる機械学習入門(第2版)』(森北出版, 2018年)
- スライドとJupyter notebook
- サポートページ

# 10.1 なぜ性能が向上するのか (1/2)

- アイディア
  - 学習データから全く独立に $L$ 個の弱識別器(誤り率 $\epsilon$, 誤りは独立)を作成
    - この仮定では、$m$ 個の弱識別器が誤る確率は二項分布 $B(m;\epsilon,L)$ に従う

$$B(m;\epsilon,L) = {}_L C_m \epsilon^m (1-\epsilon)^{L-m}$$

    - $\epsilon < 0.5$ のとき、$m > L/2$ となる $B$ は小さい値になる(本当？)




$\epsilon = 0.2, L = 11$ のときの二項分布

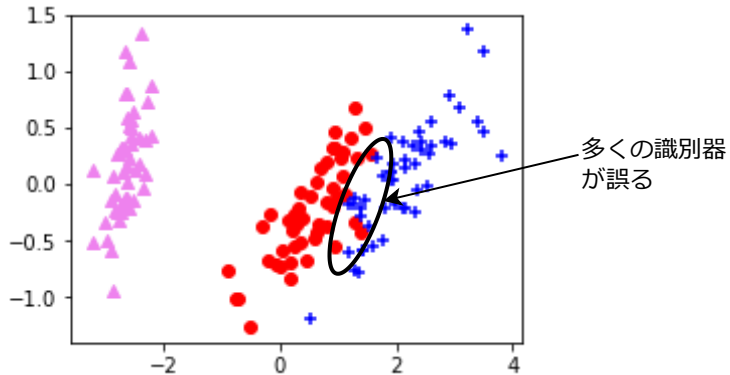
$$B(m;0.2,11) = {}_{11}C_m 0.2^m (1-0.2)^{11-m}$$

過半数$m \geq 6$ の
識別器が誤る場合

$$\sum_{m=6}^{11} B(m) = 0.01165\ldots$$

# 10.1 なぜ性能が向上するのか（2/2）

- ここまでの議論の非現実的なところ
  - 「それぞれの弱識別器の誤りが独立」→ データの誤りやすさに差はない ×



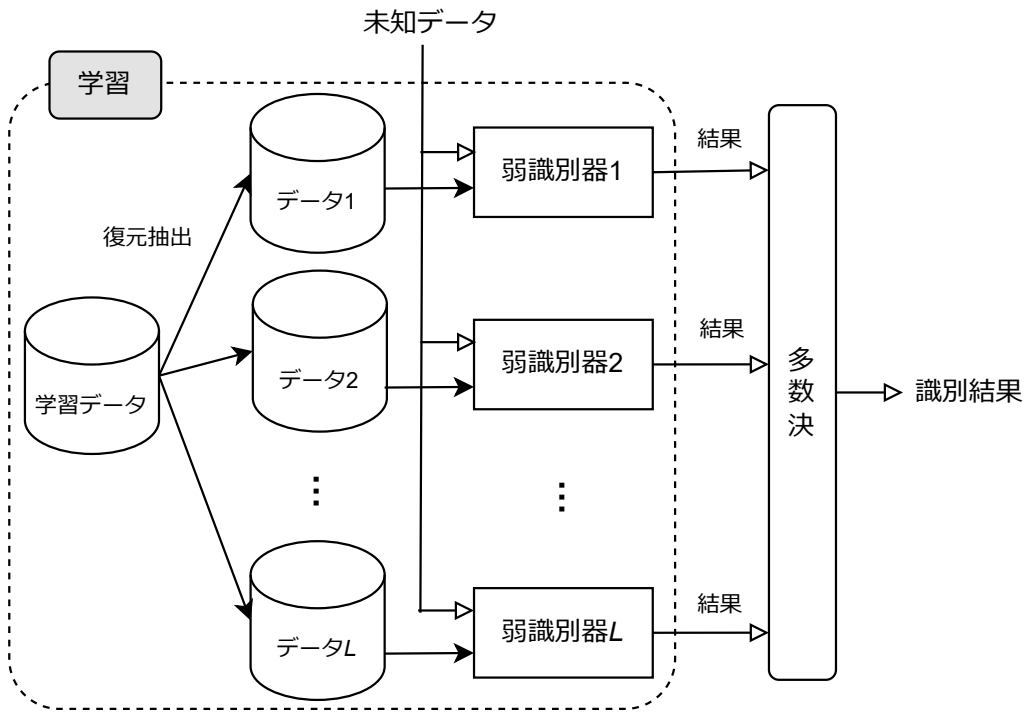

- アンサンブル学習の目標
  - 「弱識別器の誤りが独立」に近づけるために、なるべく異なる振る舞いをする弱識別器を作成する

# 10.2 バギング (1/5)

- アイディア
  - 異なる学習データから作成された弱識別器は異なる
- Bagging (bootstrap aggregating) とは
  - 元の学習データから復元抽出によって異なるデータ集合を複数作成
  - それぞれに対して弱識別器を作成
  - 結果を統合

# 10.2 バギング (2/5)

- バギングの構成



# 10.2 バギング (3/5)

- 学習
  - 学習データから復元抽出を行って、元データと同サイズのデータ集合を複数作成する
    - $N$回抽出を行って、特定のデータが抽出されない確率: $(1 - \frac{1}{N})^N$
    - $N \to \infty$ で $\frac{1}{e} \sim 0.368$
  - 各々のデータ集合に対して同じアルゴリズムで弱識別器を作成する
  - アルゴリズムは不安定(学習データの違いに敏感)なほうがよい
    - 例) 枝刈りをしない決定木
- 識別
  - 弱識別器に差を付ける根拠がないので、結果の統合は多数決

# 10.2 バギング (4/5)

- コードの例
  - 識別問題のデータ: breast_cancer 569事例
    - 特徴: 腫瘍の半径や周囲の値等の平均・標準偏差・最大値など30次元
    - クラス: 悪性(malignant) 212事例 / 良性(benign) 357事例

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn import ensemble
from sklearn.model_selection import cross_val_score

X, y = load_breast_cancer(return_X_y=True, as_frame=True)
```

# 10.2 バギング (5/5)

- コードの例
  - 識別器 `BaggingClassifier` のパラメータ
    - `base_estimator`: 弱識別器のオブジェクト。指定しなければ決定木
    - `n_estimators`: 弱識別器の数。デフォルトは10

```
clf1 = ensemble.BaggingClassifier()
scores = cross_val_score(clf1, X, y, cv=10)
print(f'{scores.mean() * 100:4.2f} +/- {scores.std() * 200:4.2f} %')
```

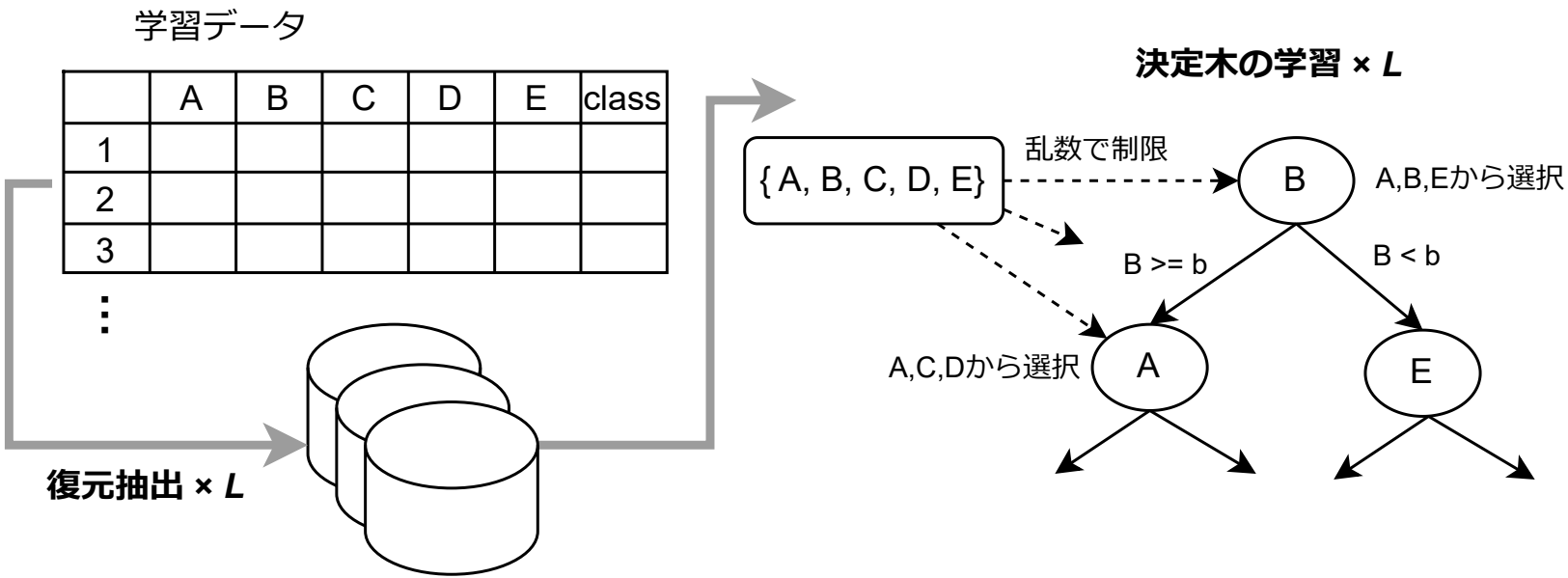
```
95.26 +/- 5.21 %
```

# 10.3 ランダムフォレスト（1/3）

- アイディア
  - バギングにおける弱識別器作成アルゴリズムにランダム性を入れることで、弱識別器間の異なりを大きくする
- ランダムフォレストとは
  - ランダム：学習の際に用いる特徴を乱数によって制限する
  - フォレスト：森 ＝ 複数の決定木
- 利点
  - 各抽出データ毎に比較的大きく異なった弱識別器ができる
  - 森のサイズ（＝決定木の数）を大きくしても、過学習が起きにくい

# 10.3 ランダムフォレスト（3/3）

- ランダムフォレストの学習



# 10.3 ランダムフォレスト (3/3)

- コードの例
  - 識別器 `RandomForestClassifier` のパラメータ
    - `max_features`: ノード選択の対象とする特徴の数。デフォルトは全特徴数の平方根
    - `n_estimators`: 弱識別器の数。デフォルトは100

```
clf2 = ensemble.RandomForestClassifier(n_estimators=10)
scores = cross_val_score(clf2, X, y, cv=10)
print(f'{scores.mean() * 100:4.2f} +/- {scores.std() * 200:4.2f} %')
```

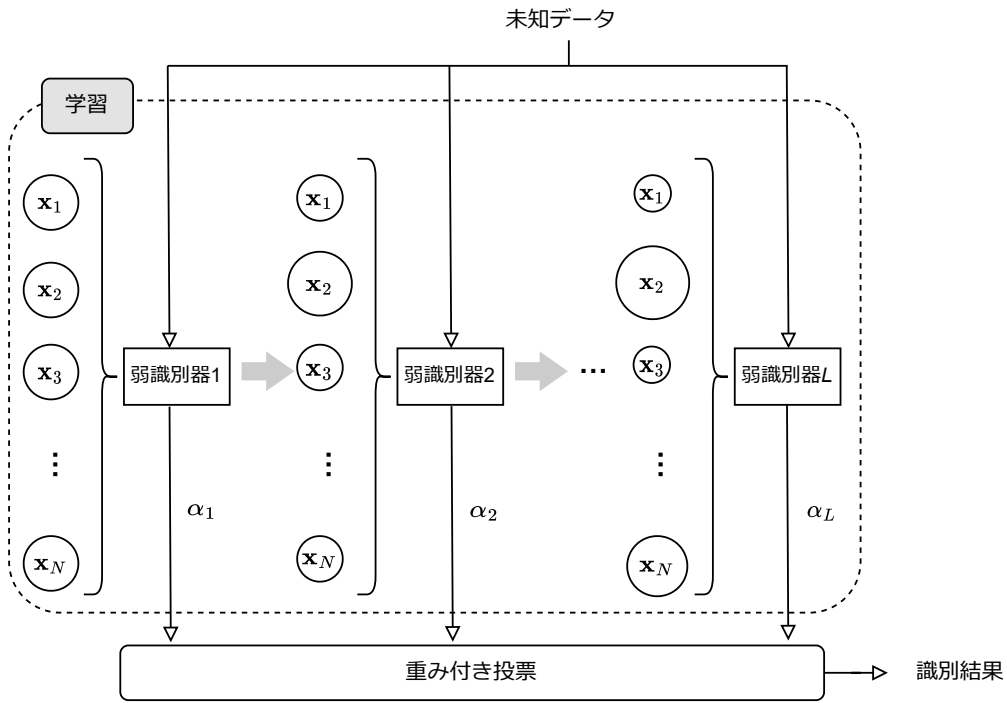
```
96.14 +/- 4.38 %
```

# 10.4 ブースティング（1/4）

- アイディア
  - 逐次的に弱識別器を追加してゆく
    - boost：（能力などを）上昇（向上）させる
  - ステップ $m$ では、ステップ $m-1$ に作成した弱識別器が誤識別を起こすデータを、より正しく識別しやすい弱識別器を作成
- 構成
  - 学習アルゴリズムは重み付きデータに対応しているものが望ましい
    - 重みに対応していない場合は、重みに比例してデータを復元抽出
  - 過学習とならないように、弱識別器として浅い決定木を用いることが多い
  - 結果は弱識別器の性能に基づく重み付き投票

# 10.4 ブースティング（2/4）

- ブースティングの構成



# 10.4 ブースティング (3/4)

- AdaBoostアルゴリズム(2値分類の場合)

1. 開始時は全学習データに同じ重みを付与
2. $m = 1$ から $L$ まで以下を繰り返し
    1. $m$ 番目の弱識別器を作成。それを用いて誤り率 $\epsilon_m$ $(< 0.5)$ を算出
    2. 誤識別されたデータの重みを $\frac{1}{2\epsilon_m}$ 倍
    3. 正しく識別されたデータの重みを $\frac{1}{2(1-\epsilon_m)}$ 倍
    4. $m$ 番目の弱識別器の重み $\alpha_m = \frac{1}{2} \ln \frac{1-\epsilon_m}{\epsilon_m}$

# 10.4 ブースティング（4/4）

- コードの例
  - 識別器 `AdaBoostClassifier` のパラメータ
    - `base_estimator`：弱識別器のオブジェクト。指定しなければ深さ1の決定木
    - `n_estimators`：弱識別器の数。デフォルトは50

```
clf3 = ensemble.AdaBoostClassifier()
scores = cross_val_score(clf3, X, y, cv=10)
print(f'{scores.mean() * 100:4.2f} +/- {scores.std() * 200:4.2f} %')
```

```
96.13 +/- 5.62 %
```

# 10.5 勾配ブースティング（1/6）

- 勾配ブースティングとは
  - ブースティングにおける各ステップで、これまでに作成したアンサンブル識別器に追加したときの損失関数の値が最小となる弱識別器を加える
- ブースティングの一般化
  - 回帰問題の場合、アンサンブル回帰器の出力は $L$ 個の弱回帰器の和で表現できる

$$F_L(\boldsymbol{x}) = \sum_{m=1}^{L} h_m(\boldsymbol{x})$$

  - 識別問題では、$\sigma(F_L(\boldsymbol{x}))$ を正例の確率とする

# 10.5 勾配ブースティング（2/6）

- 学習の手順
  - 識別器の学習を逐次加法的な学習プロセスに分解

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + h_m(\boldsymbol{x})$$

  - 第 $m$ ステップで加える弱識別器 $h_m(\boldsymbol{x})$ を選択するための損失関数 $l$ の導入
    - ステップ $m$ の時点でのアンサンブル識別器の損失を $L_m$ とする
    - 損失関数 $l$ の例
      - 回帰: 2乗誤差、Huber損失
      - 識別: log損失、指数損失

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{N} l(y_i, F_{m-1}(\boldsymbol{x}_i) + h(\boldsymbol{x}_i))$$

# 10.5 勾配ブースティング（3/6）

- 損失項の計算
  - 損失関数をTaylor展開し、$l(z) \approx l(a) + (z-a)\frac{\partial l(a)}{\partial a}$ を使って1次の項までで近似
    - $z = F_{m-1}(\boldsymbol{x}_i) + h(\boldsymbol{x}_i),\ \ a = F_{m-1}(\boldsymbol{x}_i)$ とする

$$l(y_i, F_{m-1}(\boldsymbol{x}_i) + h(\boldsymbol{x}_i)) \approx l(y_i, F_{m-1}(\boldsymbol{x}_i)) + h_m(\boldsymbol{x}_i) \left[ \frac{\partial l(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)} \right]_{F=F_{m-1}}$$

  - $F_{m-1}(\boldsymbol{x}_i)$ の値が与えられ、損失関数 $l$ が微分可能であれば、$\left[ \frac{\partial l(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)} \right]_{F=F_{m-1}}$ の値 $g_i$ は容易に計算可能
    - 例：2乗誤差なら $g_i = y_i - F_{m-1}(\boldsymbol{x}_i)$

# 10.5 勾配ブースティング（4/6）

- 弱識別器の選択
  - 深さ1の決定木なら、すべての特徴からノードを選択するだけ

$$h_m \approx \arg\min_h \sum_{i=1}^{N} h_m(\boldsymbol{x}_i) g_i$$

- 学習の高速化
  - 特徴値をヒストグラムで表現することで、分割点の探索を高速にする
  - scikit-learn で Histogram-Based Gradient Boosting として実装されている

# 10.5 勾配ブースティング（5/6）

- コードの例
  - 識別器 `GradientBoostingClassifier` のパラメータ
    - `loss`：損失関数。デフォルトは `log_loss`
    - `n_estimators`：弱識別器の数。デフォルトは100

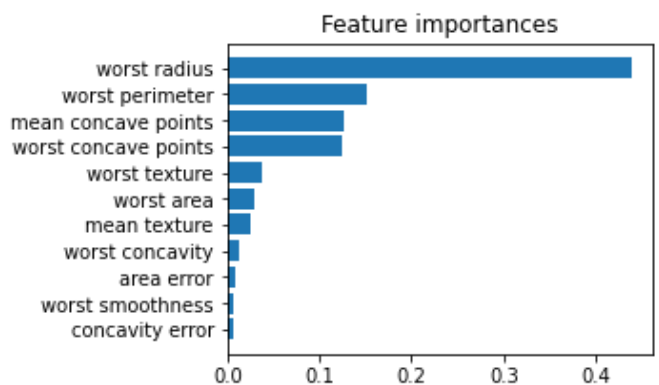
```
clf4 = ensemble.GradientBoostingClassifier()
scores = cross_val_score(clf4, X, y, cv=10)
print(f'{scores.mean() * 100:4.2f} +/- {scores.std() * 200:4.2f} %')
```

```
96.14 +/- 5.83 %
```

# 10.5 勾配ブースティング（6/6）

- 特徴の評価
  - 弱識別器が決定木の場合、Gini 不純度の減少量に基づいた特徴の評価量が得られる

```
clf4.fit(X,y)
importances = clf4.feature_importances_
indices = np.argsort(importances)[::-1]
plt.title("Feature importances")
plt.barh(bc.feature_names[indices[10::-1]], importances[indices[10::-1]])
plt.show()
```



# 10.6 まとめ

- アンサンブル学習とは
  - 弱識別器を複数組み合わせ、それらの結果を統合
  - 異なる振る舞いをする弱識別器を作る
- バギング
  - 復元抽出で異なる学習データを作成
- ランダムフォレスト
  - バギング＋決定木学習において乱数で特徴を制限
- ブースティング
  - 以前の識別器の誤りを補完する識別器を順次作成