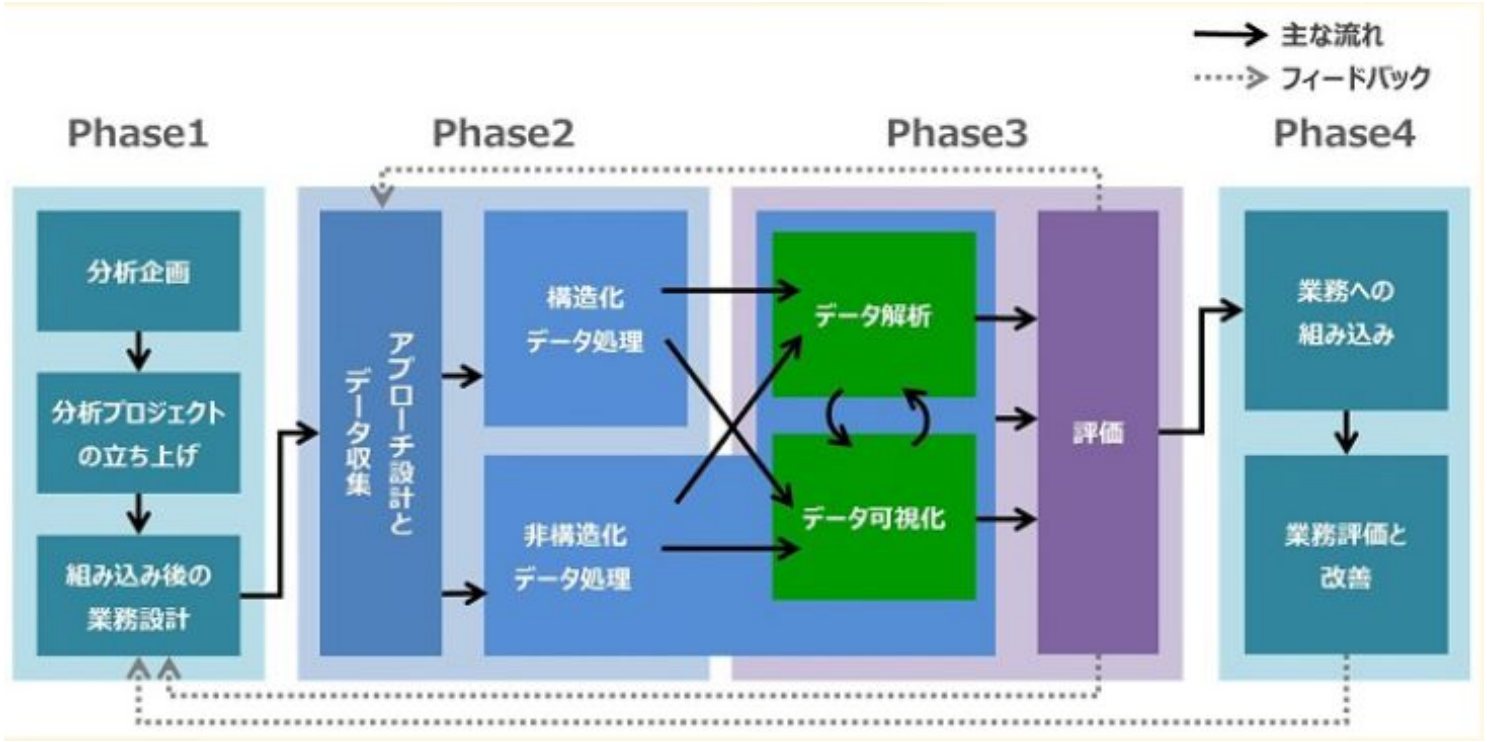
Week8 モデルの検証方法とチューニング方法

講師・スライド作成：中内

Week8で学ぶこと

1. 機械学習モデルのバリデーション方法
2. ハイパーパラメータチューニング
3. モデルを検証するための評価指標
4. 汎化性能向上のためのアンサンブル手法

データ分析プロジェクトにおけるモデル検証・評価



https://www.ipa.go.jp/jinzai/itss/itssplus.html

Week8で学ぶこと

1. 機械学習モデルのバリデーション方法
2. ハイパーパラメータチューニング
3. モデルを検証するための評価指標
4. 汎化性能向上のためのアンサンブル手法

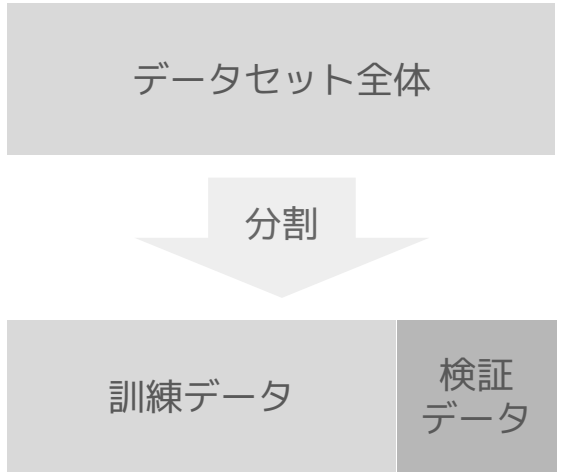
Holdout Method (ホールドアウト法)

基本的特徴

１パターンの訓練/検証データに分割してモデル評価する手法

- 最も単純なやり方
- 計算量は小さく済む
- 分布が偏るリスクがある
- 重要なクラスタが存在するデータセットには向かない

データ分割のイメージ



KFold Cross Validation（k分割交差検証法）

基本的特徴

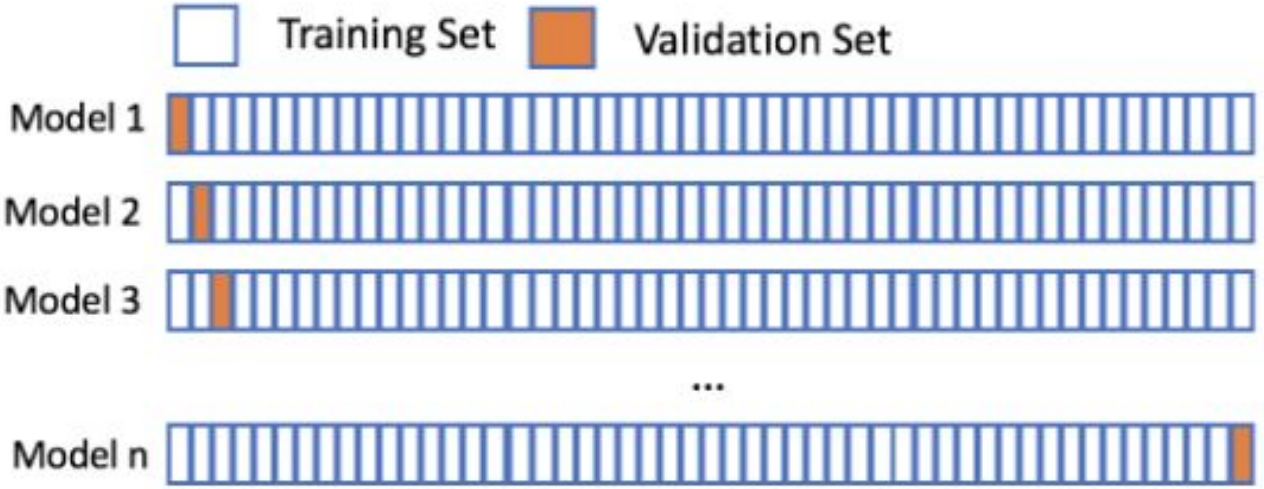
データセットをｋ個に分割し、ｋ回の検証結果から性能評価を行う手法

- CVとして最もシンプル
- Holdout法よりも過学習の見逃しを減らせる
- 複数行するのでHoldout法よりも計算コストが格段に増える

データ分割のイメージ

各回のスコアを平均などで集計

Leave-one-out Cross Validation（1個抜き交差検証）



出典：https://biol607.github.io/lectures/crossvalidation.html#31

sklearn.model_selection.**cross_val_score**

- Scikit-learnの関数を使えば簡単に実装できる
- 同じジャンルの便利な関数は他にも多数

sklearn.model_selection.**cross_val_score**

主な引数

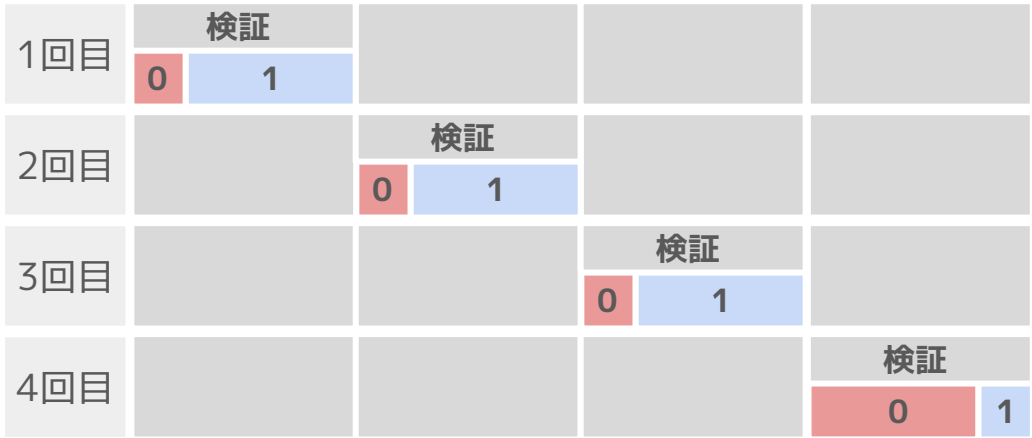
- **estimator**
  - 検証したい機械学習モデルを指定
- **X, y**
  - 説明変数と目的変数
- **cv**
  - 何分割して交差検証するか
- **scoring**（省略すると正解率）
  - 評価指標に何を使うか

使用イメージ

```
from sklearn.svm import SVC（→任意のモデル）

from sklearn.model_selection import \
    cross_val_score

scores = cross_val_score(
    estimator=SVC(), X, y, cv=4,
    scoring='roc_auc')

print(scores)
```

StratifiedKFoldの有用性 (層化抽出法を使ったK-分割交差検証法)

何も考慮せずにデータ分割した場合



各Foldに配分される目的変数の分布が偏るリスクがある

- 左図のような偏りを避ける必要がある
- 偏ると各Foldのスコア平均をとっても正しい性能把握にならない
- それを避けるため**目的変数の値が元の分布から変わらないように各Foldに配分**するのが層化抽出法
- cross_val_score関数を分類問題に使うと自動的に適用され、左図のような偏りを避けられる

データの対象期間ごとのモデルの種類

予測モデル

- 説明変数と目的変数の期間が異なる
- データの未来予測が目的

プロファイリング・モデル

- 説明変数と目的変数の期間が同じ
- データの探索的理解が目的

Week8で学ぶこと

1 機械学習モデルのバリデーション方法

2 ハイパーパラメータチューニング

3 モデルを検証するための評価指標

4 汎化性能向上のためのアンサンブル手法

モデルのハイパーパラメータをチューニングする

各機械学習モデルはハイパーパラメータによって大きく精度が変わる

- ハイパラにはたくさんの種類がある
- その組み合わせ数は膨大
- 一つひとつ手作業で検証をしていくのは作業負荷が大きい

```
tree = DecisionTreeClassifier(
    criterion='entropy',
    splitter='best'
    max_depth=3,
    min_samples_split=2,
    min_samples_leaf=1
    min_weight_fraction_leaf=0.0,
    max_features=None,
    random_state=0,
    min_impurity_decrease=0.0
```

各機械学習モデルには多数のハイパーパラメータが用意されている

ハイパーパラメータの探索(1)：グリッドサーチ

この探索手法の性質

全ての組合せを総当たり法で探索する

- 重複と漏れがない
- 探索の順番に偏りがあるので、
  - 一定回数実行しないと意義が薄い
  - 広い範囲の探索は厳しい

初手では使わない。仕上げならアリ。

試行回数	パラメータA	パラメータB	パラメータC
0 回目	0	0	0
1 回目	0	0	1
2 回目	0	0	2
3 回目	0	0	3
4 回目	0	0	4
5 回目	0	1	0
6 回目	0	1	1
7 回目	0	1	2
8 回目	0	1	3
9 回目	0	1	4
10 回目	0	2	0
11 回目	0	2	1
12 回目	0	2	2
13 回目	0	2	3
14 回目	0	2	4
15 回目	0	3	0

例えば0~4の値をとる３種のパラメータを探索するときパラメータAが0〜4まで探索されるのはB~Cのすべての探索が終わった後になってしまう

GridSearchCVの使い方

①まず、Scikit-learnパッケージから使いたいモデルが属するモジュールをインポートする。

②機械学習モデルだけでなく、グリッドサーチを行うインスタンスなど様々な種類がある

```
from sklearn.model_selection import GridSearchCV

# グリッドサーチを行うクラスインスタンスを作成
gs = GridSearchCV(estimator=SVC(),
                  param_grid=param_grid,
                  cv=5)
```

GridSearchCVの使い方

```
from sklearn.model_selection import GridSearchCV

# グリッドサーチを行うクラスインスタンスを作成
gs = GridSearchCV(estimator=SVC(),
                  param_grid=param_grid,
                  cv=5)
```

①作成するインスタンスのパラメータを引数で指定する

- **estimator** → グリッドサーチしたいモデル
- **param_grid** → 探索したいハイパラの種類と範囲を収めた辞書型のデータ
- **cv** → クロスバリデーションのk分割の数

②GridSearchCVはコンストラクタ関数なのでこれを実行した結果、左辺の変数（ここでは「gs」）に格納されるのはインスタンス。

③後続する各種処理はこのインスタンスに対して行う

インスタンスはメソッドやインスタンス変数で活用する

Step1

コンストラクタ関数で
インスタンスを作成

```
# クラスインスタンスを作成
gs = GridSearchCV(estimator=SVC(),
                  param_grid=param_grid,
                  cv=5)
```

Step2

「**.fit()**」メソッドで**グリッドサーチ**（説明変数と目的変数を渡す）

```
# データセットに対してグリッドサーチの実行
gs.fit(X_train, y_train)
```

Step3

「.best_score_」などのインスタンス変数で結果を確認！

```
# 結果データはインスタンス内に変数として蓄積されている
gs.best_score_
gs.best_params_
gs.score(X_test, y_test)
```

GridSearchCVのメソッド＆インスタンス変数

.best_score_
ベストスコアを表示

.score(X, Y)
訓練データ以外のデータでの精度を表示

.best_params_
最適なハイパラを表示

メソッドやインスタンス変数の使い方がだいじ！

Scikit-learnの公式ドキュメントをみると思わぬ便利機能を知れることがある
→機能は随時更新されるので積極的に公式情報をキャッチアップしよう！

ハイパーパラメータの探索(2)：ランダムサーチ

ランダムサーチ

無作為にパラメータを探索する

- 重複と漏れが生じる
- 探索の順番がランダムなので、
  - 広い範囲を満遍なく探索できる
  - 途中で止めても一定の意義がある

仕上げには向かない。初手ならアリ。

グリッドサーチ

全ての組合せを総当たり法で探索する

- 重複と漏れがない
- 探索の順番に偏りがあるので、
  - 広い範囲の探索は厳しい
  - 一定回数実行しないと意義が薄い

初手では使わない。仕上げならアリ。

sklearn.model_selection.**RandomizedSearchCV**

主な引数

- **estimator**
  - ハイパラを探索する機械学習モデル
- **param_distributions**
  - ハイパラの種類と探索範囲を収めた辞書
- **n_iter**
  - 探索回数
- **scoring**
  - バリデーションに使う評価指標

【ほとんどGridSearchCVと同じ！】

使用イメージ

```
from sklearn.model_selection import \
    RandomizedSearchCV

rs = RandomizedSearchCV(
    estimator=SVC(),
    param_distributions=params,
    n_iter=500, scoring='roc_auc',
    random_state=0)

rs.fit(X_train, y_train) # 学習
print(rs.best_score_) # 結果表示
print(rs.best_params_)
```

その他のハイパラ探索の手法：ベイズ最適化

- ベイズ最適化を使ったアルゴリズムによる自動探索
- 過去の探索履歴を考慮して、次に探索すべきハイパラを合理的に選択する

Week8で学ぶこと

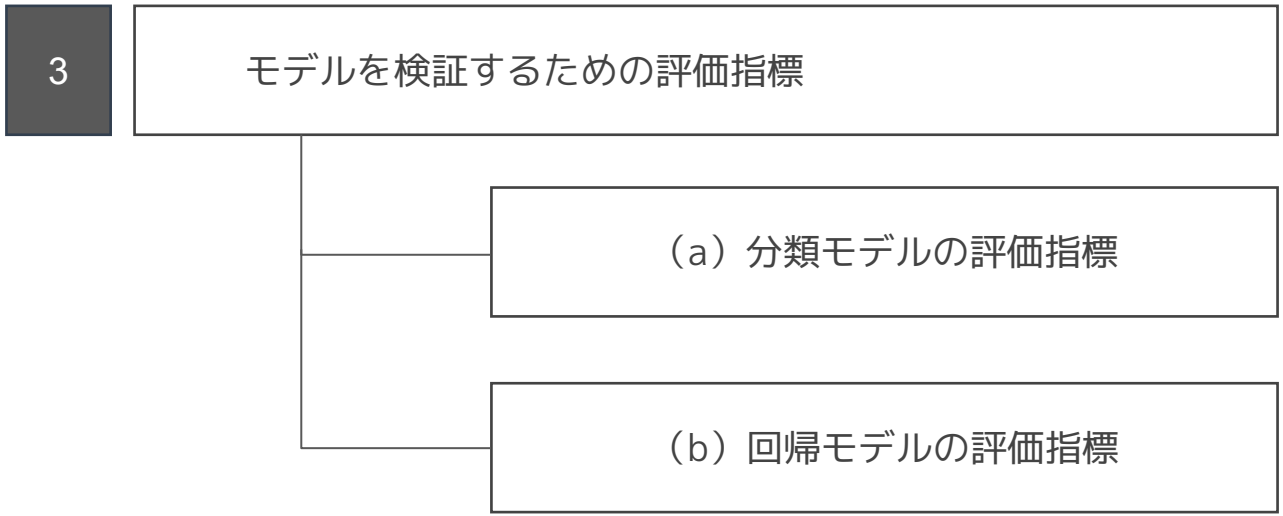
1 機械学習モデルのバリデーション方法

2 ハイパーパラメータチューニング

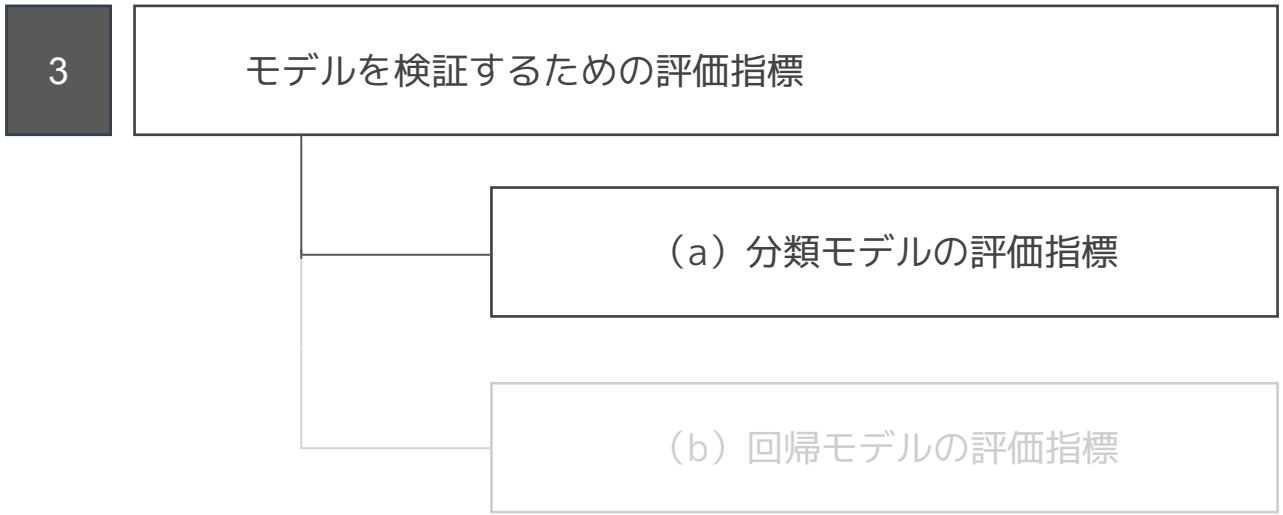
3 モデルを検証するための評価指標

4 汎化性能向上のためのアンサンブル手法

Week8で学ぶこと



Week8で学ぶこと



混同行列（Confusion Matrix）

		予測値	
		負例/陰性/ Negative/0	正例/陽性/ Positive/1
正解値	負例/陰性/ Negative/0	TN True Negative 真陰性	FP False Positive 偽陽性
	正例/陽性/ Positive/1	FN False Negative 偽陰性	TP True Positive 真陽性

混同行列の観点が必要な理由

例：正解率99%の病気診断システムができたぞ！

→混同行列で検証するとどうなるか？

混同行列の観点が必要な理由

例：正解率99%の病気診断システムができたぞ！

正解値	負例/陰性/ Negative/0	10,090件
	正例/陽性/ Positive/1	10件

混同行列の観点が必要な理由

例：正解率99%の病気診断システムができたぞ！

		予測値	
		負例/陰性/ Negative/0	正例/陽性/ Positive/1
正解値	負例/陰性/ Negative/0	True Negative 10,000件	False Positive 90件
	正例/陽性/ Positive/1	False Negative 8件	True Positive 2件

Accuracy（正解率）$= \frac{TP + TN}{TP + TN + FP + FN}$

正解値の**偏り**に関係なく、予測値と正解値が一致していた割合

		予測値	
		負例/陰性/ Negative/0	正例/陽性/ Positive/1
正解値	負例/陰性/ Negative/0	True Negative 10,000件	False Positive 90件
	正例/陽性/ Positive/1	False Negative 8件	True Positive 2件

(2 + 10,000) / (2 + 10,000 + 90 + 8) = **正解率99.0%**

→そもそも正解値が負例に偏っているデータセットにおいては、
適当にほぼ全て負例と予測してしまっても正解率が高く出てしまう

Recall（再現率）$= \frac{TP}{TP + FN}$

真に正例であるもの(TP+FP)のうち、正しく正例と予測した件数(TP)の割合

		予測値	
		負例/陰性/ Negative/0	正例/陽性/ Positive/1
正解値	負例/陰性/ Negative/0	True Negative 10,000件	False Positive 90件
	正例/陽性/ Positive/1	False Negative **8件**	True Positive **2件**

2 / (2 + 8) = **再現率20.0%**

「見逃しが許されない事象を取りこぼしなく発見しうるか」をみる指標

※精度と訳すこともある

Precision（適合率）$= \frac{TP}{TP + FP}$

正例と予測した件数(TP + FP)のうち、実際に正例だった件数(TP)の割合

		予測値	
		負例/陰性/ Negative/0	正例/陽性/ Positive/1
正解値	負例/陰性/ Negative/0	True Negative 10,000件	False Positive **90件**
	正例/陽性/ Positive/1	False Negative 8件	True Positive **2件**

2 / (2 + 90) = **適合率2.2%**

正例に対する施策実行がハイコストである場合などに
予測結果どおり行動するコストパフォーマンスをみる指標

F-Value（F1値）

適合率と再現率の「調和平均」

$$F_1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}}$$

$$= \frac{2 \cdot recall \cdot precision}{recall + precision}$$

$$= \frac{2TP}{2TP + FP + FN}$$

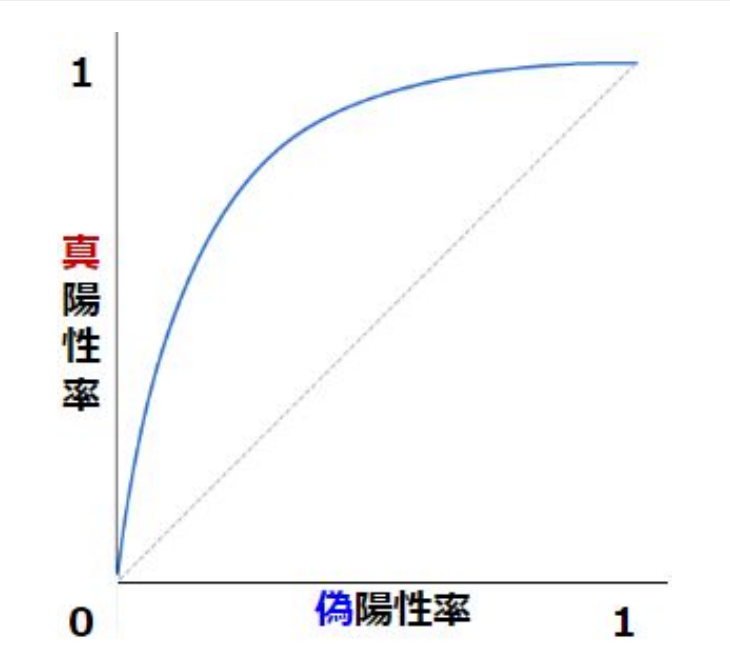
		予測値	
		負	正
正解値	負	TN 10,000件	FP 90件
	正	FN 8件	TP 2件

F1値＝0.04%

適合率と再現率の両方の観点から
機械学習モデルを比較評価したい時に使う指標

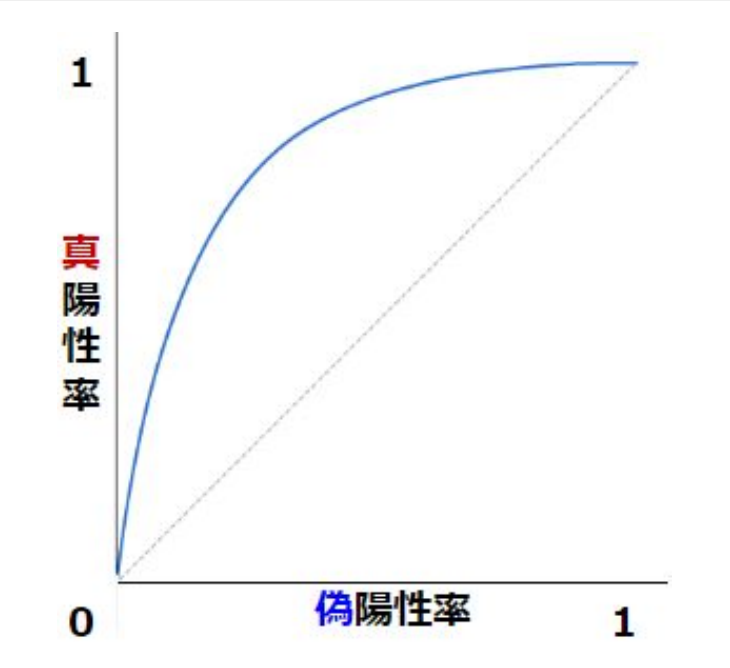
Receiver **O**perating **C**haracteristic - **A**rea **U**nder the **C**urve

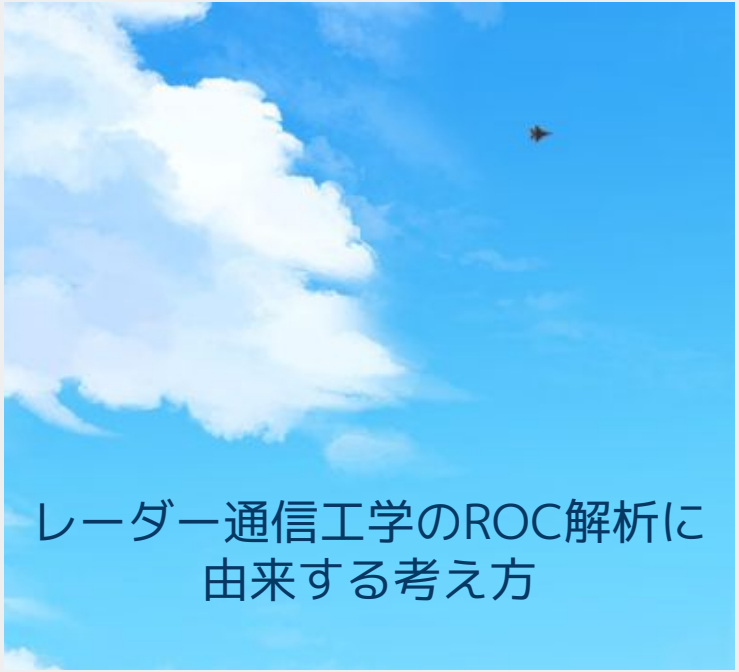
ROC曲線とROC-AUCスコア



Receiver **O**perating **C**haracteristic - **A**rea **U**nder the **C**urve

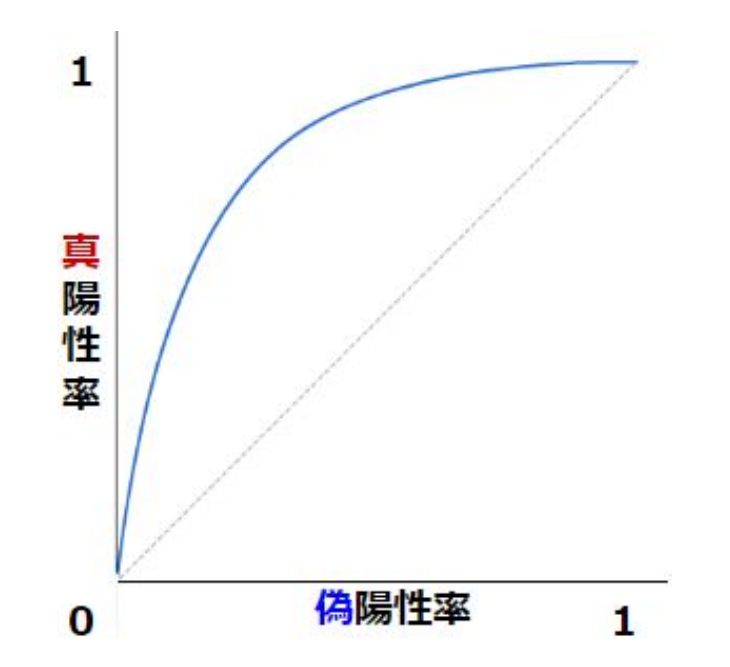
ROC曲線とROC-AUCスコア





Receiver **O**perating **C**haracteristic - **A**rea **U**nder the **C**urve

ROC曲線とROC-AUCスコア

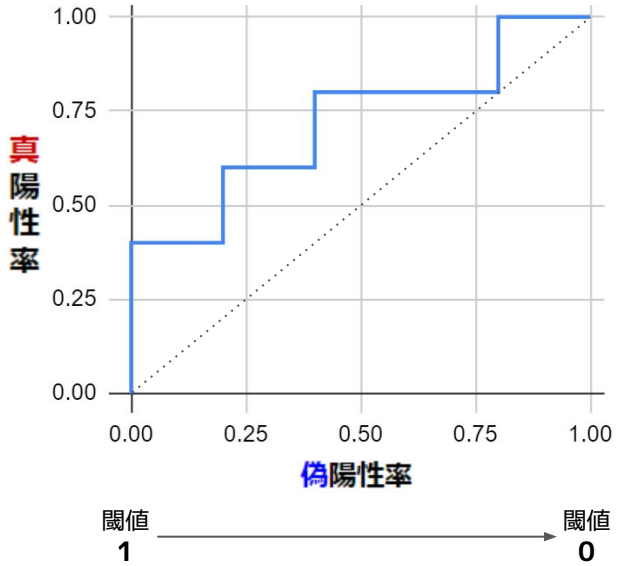


二値分類でも予測値を確率値で出せる

```
# 二値分類モデルの作成
clf = DecisionTreeClassifier()
clf.fit(X_train,y_train)  # 学習
y_pred = clf.predict_proba(X_test) # 予測

print(y_pred)  # 陰性と陽性の各確率値が出る
>>[[0.090 0.910]
   [0.696 0.304]
    ...
```

Receiver **O**perating **C**haracteristic - **A**rea **U**nder the **C**urve

ROC曲線とROC-AUCスコア



ID	正解	予測	閾値ごとの判定				
			0.9	0.8	0.5	0.2	0.1
001	1	0.95	1	1	1	1	1
002	1	0.91	1	1	1	1	1
003	0	0.83	0	1	1	1	1
004	1	0.75	0	0	1	1	1
005	0	0.61	0	0	1	1	1
006	1	0.52	0	0	1	1	1
007	0	0.44	0	0	0	1	1
008	0	0.35	0	0	0	1	1
009	1	0.29	0	0	0	1	1
010	0	0.11	0	0	0	0	1

sklearn.metrics.**roc_auc_score**

主な引数

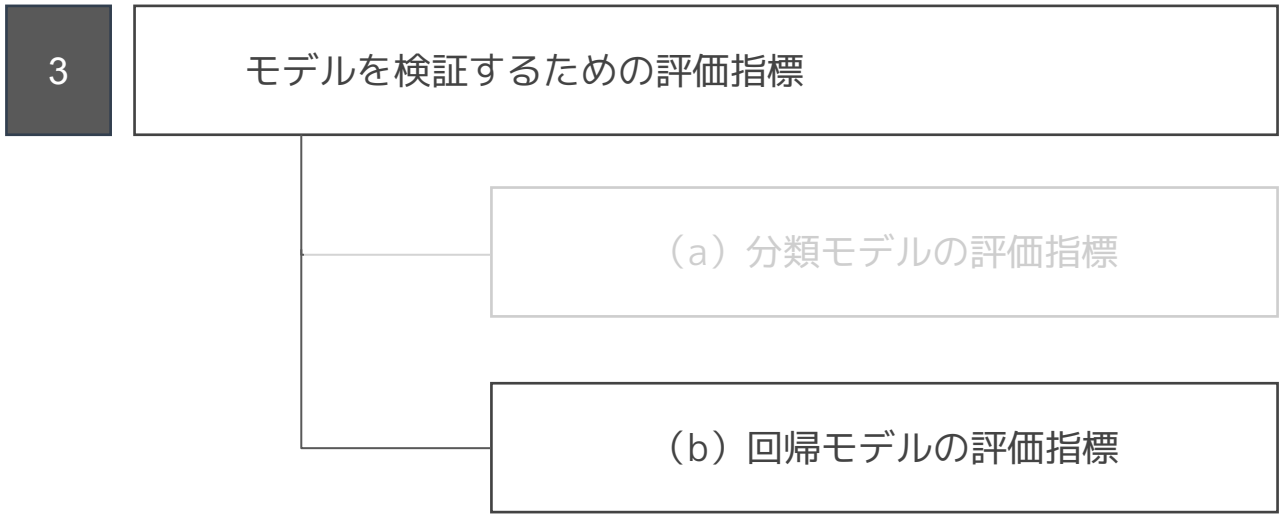
- **第一引数**
  - 正解値の格納された変数を渡す
- **第二引数**
  - 予測値の格納された変数を渡す

使用イメージ

```
from sklearn.metrics import roc_auc_score

model = SVC(probability=True)
model.fit(X_train, y_train)
y_pred = model.predict_proba(X_test)

            正解値     予測値
roc_auc_score(y_test, y_pred)

>> 0.865544
```

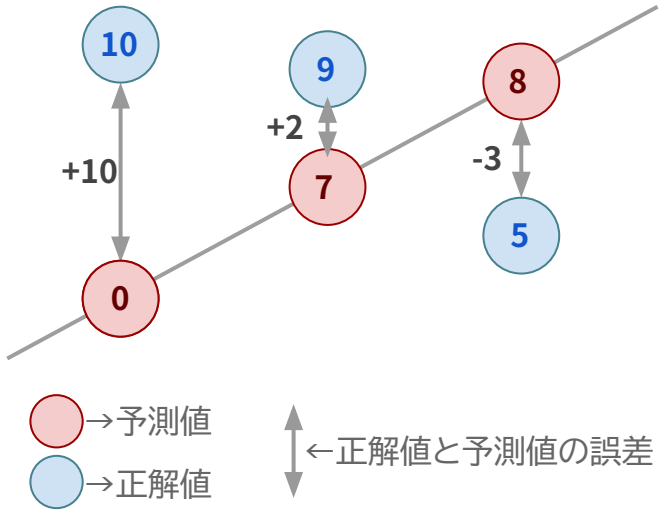
正解率を算出するaccuracy_score関数などと全く同じ使い方で簡単に算出できる

Week8で学ぶこと



回帰モデルの評価指標の基本発想

- **誤差**（=正解値－予測値）の平均値を知りたい
- 誤差は正だったり負だったりするので、**単純に足し上げると誤差同士相殺しあってしまい**正しく計算できない
- 正でも負でも誤差には変わりないので、**この余計な正or負の違いをどう処理するか**
- その違いで指標の種類が分かれる

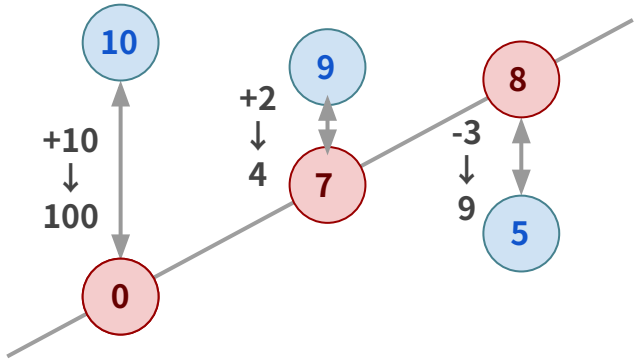


回帰モデルの評価指標（平均二乗誤差）

MSE：Mean Squared Error

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

- 誤差が大きいほど過大に評価する
- 外れ値に過敏に反応する
- 元のデータからは単位が変わってしまう



$$MSE = \frac{1}{3}\cdot(100 + 4 + 9) = 37.7$$

→予測値

→正解値

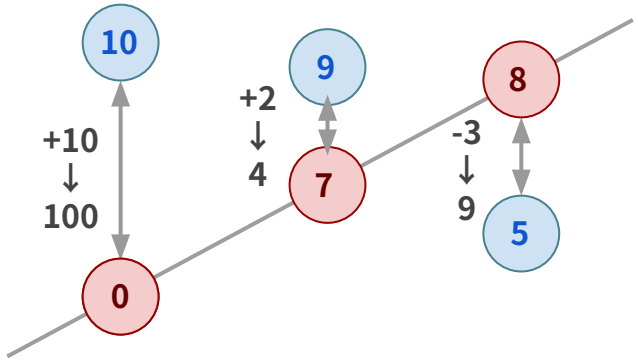
←正解値と予測値の誤差

回帰モデルの評価指標（平均平方二乗誤差）

RMSE：Root Mean Squared Error

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

- MSEの平方根をとって元のデータの単位に戻したもの
- MSEより直感的に理解しやすい
- 基本性質はMSEと同じ



$$RMSE = \sqrt{37.7} = 6.1$$

→予測値

→正解値

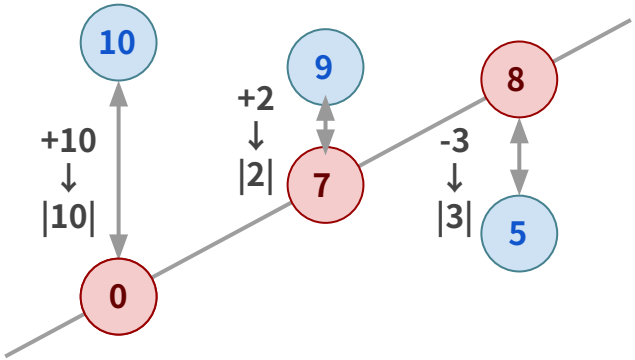
←正解値と予測値の誤差

回帰モデルの評価指標（平均絶対誤差）

MAE：Mean Absolute Error

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

- 元のデータから単位を変えないので直感的に理解しやすい
- 誤差の大きさを過大評価しない
- RMSEより外れ値の影響を受けにくい



$$MAE = \frac{1}{3}\cdot(|10| + |2| + |3|) = |5|$$

→予測値

→正解値

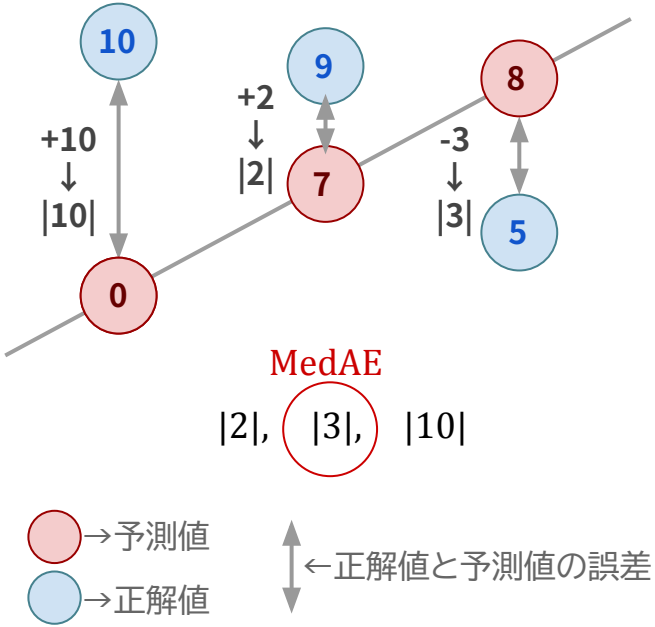
←正解値と予測値の誤差

回帰モデルの評価指標（中央絶対誤差）

MedAE：Median Absolute Error

$$MedAE(y, \hat{y}) = median(|y_1 - \hat{y}_1|, ..., |y_n - \hat{y}_n|)$$

- 平均の代わりに中央値を使って絶対誤差を算出する評価指標
- 誤差平均が歪むレベルの外れ値があっても影響を受けない
- 頻繁にみかける指標ではない

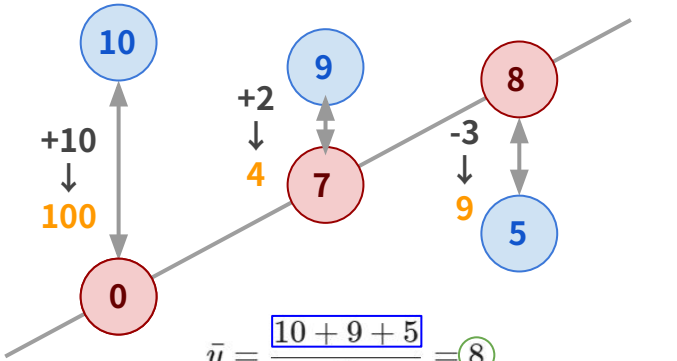


回帰モデルの評価指標（決定係数R2）

R2：R squared

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y}_i)^2} \qquad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

- 回帰式の当てはまり度合いを表す
- 最大値は１
- 誤差を二乗しているので、外れ値への敏感さはMSEと同じ
- 正解の平均を予測値とするのと比べて誤差を減らせているかをみるもの



$$\bar{y} = \frac{10+9+5}{3} = 8$$

$$R^2 = 1 - \frac{100+4+9}{(10-8)^2+(9-8)^2+(5-8)^2} = -7.07$$

→予測値

→正解値

←正解値と予測値の誤差

回帰系のスコアを算出する関数

主な引数

- **第一引数**
  - 正解値の格納された変数を渡す
- **第二引数**
  - 予測値の格納された変数を渡す

```
from sklearn.metrics import \
    mean_squared_error,  # MSE(平均二乗誤差)
    mean_absolute_error,  # MAE(平均絶対誤差)
    median_absolute_error,  # MedAE(中央絶対誤差 )
    r2_score  # R2(決定係数)

# 別途予測値を算出しておく
y_pred = model.predict(X_test)

                正解値     予測値
mean_squared_error(y_test, y_pred)

>> 37.7
```

使い方自体はroc_auc_score関数やaccuracy_score関数などと全く同じ

Week8で学ぶこと

1 機械学習モデルのバリデーション方法

2 ハイパーパラメータチューニング

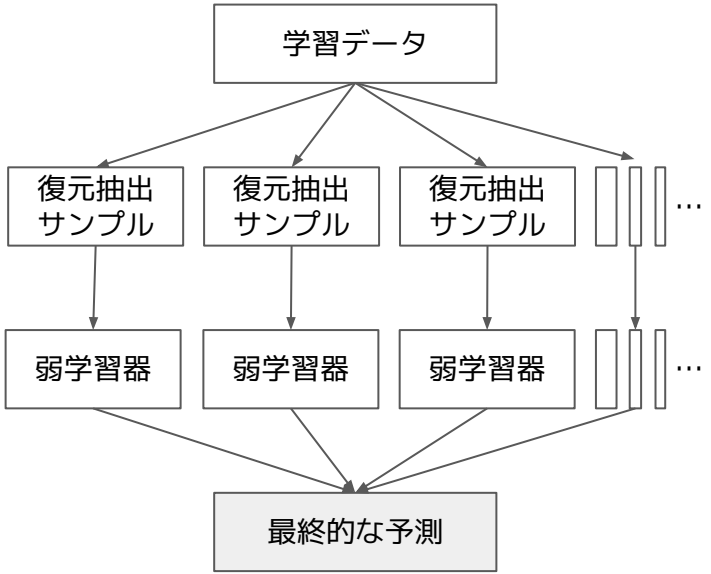
3 モデルを検証するための評価指標

4 汎化性能向上のためのアンサンブル手法

バギング

概　要

- 1996年にBreimanによって提案
- ブートストラップ法によって複数の「弱学習器」を構築
- その結果を集約して最終的な予測値を作成
- 結果集約は分類なら多数決、回帰なら平均値など



sklearn.ensemble.**Bagging**Classifier/Regressor

主な引数

- **base_estimator**
  - バギングしたい機械学習モデル
- **n_estimator**
  - 何個のモデルでバギングするか
- **random_state**
  - 再現性確保のための値

使用イメージ

```
from sklearn.ensemble import BaggingClassifier
from sklearn.svm import SVC（→ここは何でもOK）

clf = BaggingClassifier(
    base_estimator=SVC(),
    n_estimators=100,
    random_state=0)

clf.fit(X_train, y_train) # 学習
y_pred = clf.predict(X_test) # 予測
```

sklearn.ensemble.**Bagging**Classifier/Regressor

さらに使いこなすための引数

- **max_samples**：*float(0.0〜1.0)*
  - 何割のレコードを抽出するか
- **max_features**：*float(0.0〜1.0)*
  - 何割の説明変数を抽出するか
- その他、便利なものが多数
  - 公式ドキュメントをみて使いこなす
    https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

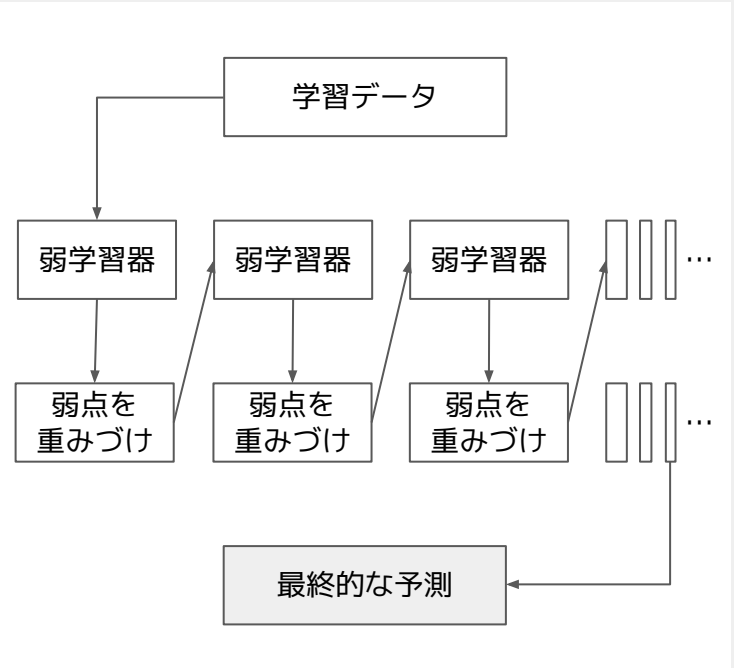
主なメソッド

- **.fit**(*X, y*)
  - 訓練の実行
- **.predict**(*X*)
  - 予測の実行
- **.score**(*X, y*)
  - 正解率(ACC)の表示

ブースティング

概　要

- 元の学習データを一定回数抽出して弱学習器を作成
- 弱学習器の予測誤りと、正解サンプルを比較して、その誤り率と重要度を元に次の弱学習器をつくる
- 計算毎に全体の重みを再調整する
- これを繰り返し、最終予測を作成



sklearn.ensemble.**AdaBoost**Classifier/Regressor

主な引数

- **base_estimator**
  - ブースティングしたいモデル
- **n_estimator**
  - 何回ブースティングするか
- **random_state**
  - 再現性確保のための値

使用イメージ

```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.svm import SVR（→ここは何でもOK）

reg = AdaBoostRegressor(
    base_estimator=SVR(),
    n_estimators=100,
    random_state=0)

reg.fit(X_train, y_train) # 学習
y_pred = reg.predict(X_test) # 予測
```

sklearn.ensemble.**RandomForest**Classifier/Regressor
sklearn.ensemble.**GradientBoosting**Classifier/Regressor

主な引数

- **n_estimator**
  - 何個でアンサンブルするか
- **random_state**
  - 再現性確保のための値
- その他多数のハイパラ

使用イメージ

```
from sklearn.ensemble import \
    GradientBoostingRegressor

reg = GradientBoostingRegressor(
    random_state=0)

reg.fit(X_train, y_train) # 学習
y_pred = reg.predict(X_test) # 予測
```

初めからアルゴリズム内にアンサンブル処理が組み込まれているので
使い方の面では単体モデルと全く同じ使い方ができる。

xgboost.**XGB**Classifier/Regressor

主な引数

- **n_estimator**
  - 何回ブースティングするか
- **random_state**
  - 再現性確保のための値
- その他多数のハイパラ

使用イメージ

```
from xgboost import XGBRegressor

reg = XGBRegressor(n_estimators=100,
                   random_state=0)

reg.fit(X_train, y_train) # 学習
y_pred = reg.predict(X_test) # 予測
```

勾配ブースティングはsklearn以外にもライブラリが公開されている（**LightGBM**, **CatBoost**など）

sklearnと共通の使い方ができるように配慮されているので、簡単に使うことができる

early_stopping_roundsを活用して過学習を避ける

検証スコアが指定したラウンド数改善されない場合、学習を早期終了する設定ができる

- **eval_set**
  - バリデーションを行うデータセットを指定
- **eval_metric**
  - バリデーションに用いる評価指標を指定する
- **early_stopping_rounds**
  - 改善が何回なければ早期終了するか

```
from xgboost import XGBRegressor

X_train, X_valid, y_train, y_valid = \
    train_test_split(X, y, test_size=0.2)

reg = XGBRegressor(n_estimators=1000)

reg.fit(X_train, y_train,
        eval_set=[(X_valid, y_valid)],
        eval_metric='rmse',
        early_stopping_rounds=20)

y_pred = reg.predict(X_test) # 予測
```

XGBには２種類あるので混同に気をつける

本体APIの使用イメージ

```
import xgboost as xgb

dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test)

params = {"objective" : "reg:squarederror",
          "eval_metric" : "rmse"}

reg = xgb.train(params, dtrain,
                num_boost_round=100)
y_pred = reg.predict(dtest)
```

Scikit-LearnAPIの使用イメージ

```
from xgboost import XGBRegressor

reg = XGBRegressor(n_estimators=100,
                   random_state=0)

reg.fit(X_train, y_train) # 学習
y_pred = reg.predict(X_test) # 予測
```