

# GCI 第9回 モデルの検証方法とチューニング方法

---



松尾・岩澤研究室  
MATSUO-IWASAWA LAB UTOKYO

講師・スライド作成：中内

2024/12/3

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

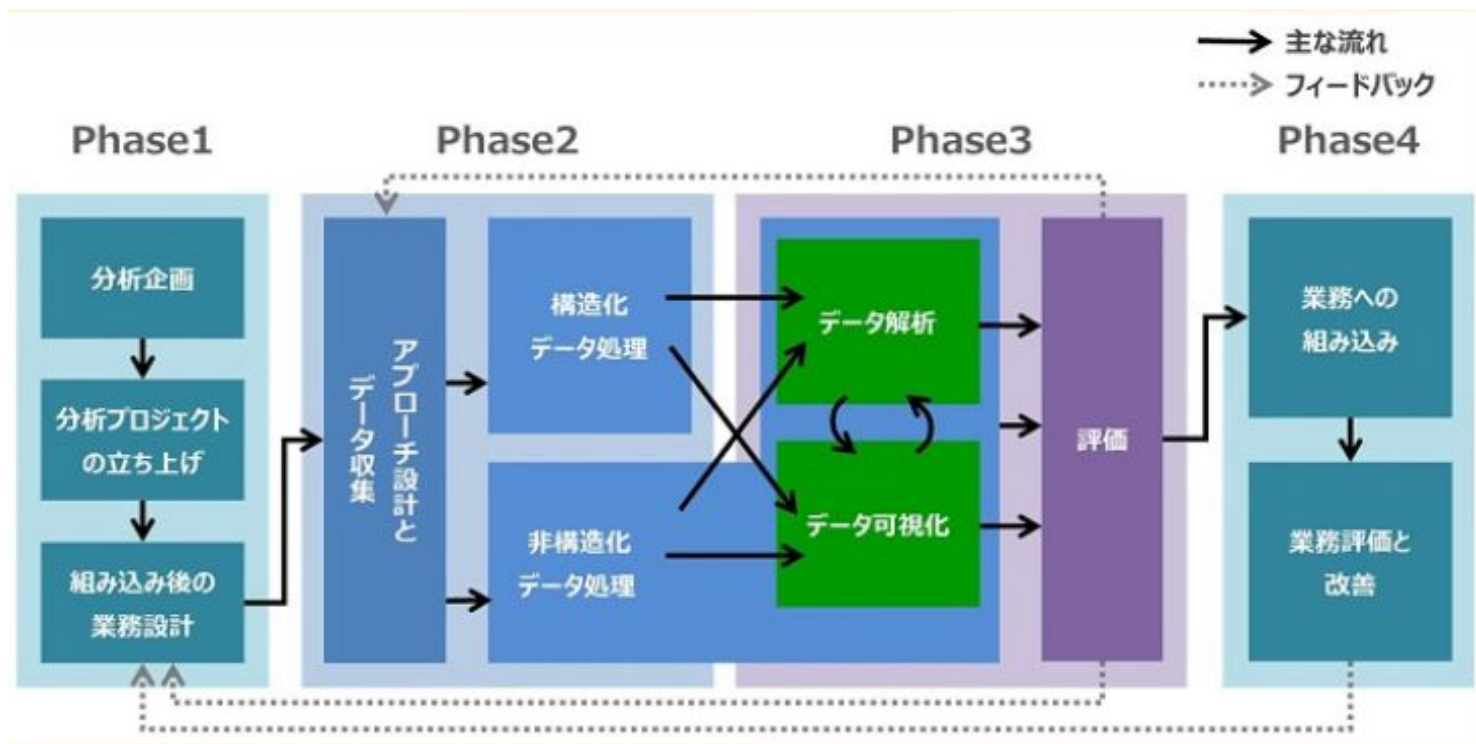
3

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/skill-standard/plus-it-ui/itssplus/data\\_science.html](https://www.ipa.go.jp/jinzai/skill-standard/plus-it-ui/itssplus/data_science.html)

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

# 復習：予測値を出すまでのミニマムな流れ



モデルをインポート

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

機械学習の3ステップ！

```
model = SVC() # インスタンスを作成
```

```
model.fit(X_train, y_train) # 学習
```

```
y_pred = model.predict(X_test) # 予測
```

# 復習：訓練用データとテスト用データの用意



```
df = pd.read_csv("csvを保存したファイルパス")
```

CSVをDataFrameで読み込む

```
X = df[["説明変数"]] # 説明変数を入れた変数
```

```
y = df["目的変数"] # 目的変数を入れた変数
```

説明変数と目的変数を用意する

```
from sklearn.model_selection import \
    train_test_split
```

```
X_train, X_test, y_train, y_test = \
    train_test_split(X, y)
```

訓練用データと検証用データに分割

# バリデーション (Validation) とは



## 機械学習モデルの性能を評価するプロセスのこと

```
df = pd.read_csv("csvを保存したファイルパス")
```

```
X = df[["説明変数"]] # 説明変数を入れた変数
```

```
y = df["目的変数"] # 目的変数を入れた変数
```

```
from sklearn.model_selection import \
    train_test_split
```

```
X_train, X_test, y_train, y_test = \
    train_test_split(X, y)
```

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

```
model = SVC() # インスタンスを作成
```

```
model.fit(X_train, y_train) # 学習
```

```
y_pred = model.predict(X_test) # 予測
```

```
print(model.score(X_test, y_test))
```

```
>> 0.733 # スコア
```

## 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





## 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

1回目	検証			
2回目		検証		
3回目			検証	
4回目				検証

各回のスコアを平均などで集計

- 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



## 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)
```

## 何も考慮せずにデータ分割した場合

1回目	検証 0 1			
2回目		検証 0 1		
3回目			検証 0 1	
4回目				検証 0 1

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

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

# ハイパーパラメータ（ハイパラ）とは




## 機械学習モデルの学習プロセスや構造を制御する設定値のこと

インスタンス作成時に何も渡さなくてもOK

```
model = SVC()
```

でも色々な設定値を渡すこともできる

```
model = SVC(  
    C=1.0,  
    kernel='rbf',  
    random_state=0  
)
```



この設定値を**ハイパーパラメータ**と呼ぶ

長いので「ハイパラ」とも略します

これまでの教材でも度々ハイパラが登場済みです

決定木の例

```
model = DecisionTreeClassifier(  
    criterion='entropy',  
    max_depth=5,  
    random_state=0  
)
```

KNNの例

```
model = KNeighborsClassifier(  
    n_neighbors=15,  
    leaf_size=30,  
    random_state=0  
)
```

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

## 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

- 利点：重複と漏れがない
- 欠点：ハイパラの組合せ数に対して計算量の増加幅が巨大なため↓
  - 実行に時間がかかる
  - 幅広い組合せの探索ができない

初手では使わない。仕上げには向く

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

グリッドサーチは全ての組合せを順番にしらみつぶしに探索していく手法



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クラ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)
```

```
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はクラス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

クラスを実行して  
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)
```

**.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！

## 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。初手には向く



## 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。仕上げには向く

## 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_)
```

- 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



O P T U N A



HYPEROPT

## グリッドサーチ

- ハイパラ探索の仕上げ作業に使う

## ランダムサーチ

- ハイパラ探索の初手で広範囲を探索する際に使う

## ベイズ最適化

- 高度にオートマチックにハイパラ調整したいときに使う
- 初心者にとっては上記二つより学習コストが若干高め



ハイパラ探索のスキルよりもEDAやFE（特徴量エンジニアリング）のスキルの方が明白に重要度が高いです。コンペではついお手軽に取り組めるハイパラ調整に逃げ込みたくなりますが、精度改善の本筋はEDAとFEであることを念頭に置いたうえで学習時間の配分を考えるのをお勧めします（初心者が今むりにベイズ最適化に取り組む必然性は薄いです。）



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

3

モデルを検証するための評価指標

(a) 分類モデルの評価指標

(b) 回帰モデルの評価指標

3

モデルを検証するための評価指標

(a) 分類モデルの評価指標

(b) 回帰モデルの評価指標

# 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

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？

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

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

$$(2 + 10,000) / (2 + 10,000 + 90 + 8) = \text{正解率99.0\%}$$

→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してしまう



# Recall（再現率）

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

真に正例であるもの(TP+FN)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 <b>8件</b>	True Positive <b>2件</b>

$$2 / (2 + 8) = \text{再現率}20.0\%$$

「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

# Precision (適合率) ※精度と訳すこともある



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

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 <b>90件</b>
	正例/陽性/ Positive/1	False Negative 8件	True Positive <b>2件</b>

$$2 / (2 + 90) = \text{適合率}2.2\%$$

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

## 適合率と再現率の「調和平均」

$$\begin{aligned} F_1 &= \frac{2}{\frac{1}{recall} + \frac{1}{precision}} \\ &= \frac{2 \cdot recall \cdot precision}{recall + precision} \\ &= \frac{2TP}{2TP + FP + FN} \end{aligned}$$

		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

# 機械学習モデルの予測値を確率値で取得する



**predict\_proba** メソッド：各レコードの陰性と陽性の予測確率を格納したnumpy.ndarrayを出力する

## 使い方

```
# 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

    陰性の確率  陽性の確率
>>[[0.090  0.910]
    [0.696  0.304]
    ...]
```

## いつものpredict()の場合

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

print(y_pred) # 陰性=0 陽性=1の二値で出てくる
>>[1, 0, 1, 0, 1, 1, 1, 0, ...]
```

# 機械学習モデルの予測値を確率値で取得する



**predict\_proba** メソッド：各レコードの陰性と陽性の予測確率を格納したnumpy.ndarrayを出力する

## axis=1方向にindex=1を指定して使う

```
# 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[:,1]) # 大抵はindex指定して使う
>>[0.910, 0.304, 0.603, 0.122, 0.730, ...]
```

※ axis=1方向にindex指定すると実質的に  
「陽性の確率は何 %か」を格納した配列として使える

## いつものpredict()の場合

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

print(y_pred) # 陰性=0 陽性=1の二値で出てくる
>>[1, 0, 1, 0, 1, 1, 1, 0, ...]
```

※ この値はpredict\_probaの値を四捨五入しただけ  
の値になっている

# 陽性判定の「閾値」の考え方



**陽性とする閾値** ... 確率予測がいくつ以上なら陽性と判断するかを定める値

## 確率予測値

```
# 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[:,1])
>>[0.910, 0.304, 0.603, 0.122, 0.730, ...]
```

## 閾値によって結果は変わる

予測値	閾値0.8	閾値0.7	閾値0.5	閾値0.3
0.910	1	1	1	1
0.304	0	0	0	1
0.603	0	0	1	1
0.122	0	0	0	0
0.730	0	1	1	1

# この2つのモデルは同じ評価で良い？



四捨五入後は同じ正解率でも**四捨五入前の値に存在する差**もきちんと評価した方が良いケースがある

四捨五入**前**の予測確率

idx	正解	モデル1	モデル2
01	1	0.99	0.51
02	1	0.98	0.53
03	0	0.02	0.49
04	0	0.01	0.47
05	1	0.97	0.52

四捨五入**後**

idx	正解	モデル1	モデル2
01	1	1	1
02	1	1	1
03	0	0	0
04	0	0	0
05	1	1	1

分類モデルが出力する予測確率と実際のラベルとの間の**誤差**を測定する関数

- 使い方は他の評価系関数と同じ
- **正解値**と**予測値**を引数に渡すだけ

- スコアが小さいほど良い
- 正解率と違って解釈できる値ではない
- モデル間比較するための相対値

## 使用イメージ

```
from sklearn.metrics import log_loss

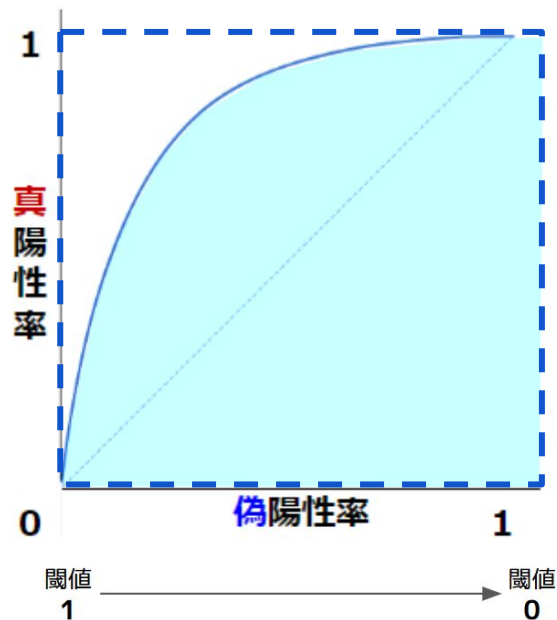
model = SVC(probability=True)
model.fit(X_train, y_train)
y_pred = model.predict_proba(X_test)[: , 1]

      正解値      予測値
log_loss(y_test, y_pred)

>> 0.826871
```

※ loglossを変形した関数は深層学習の技術でも使われている

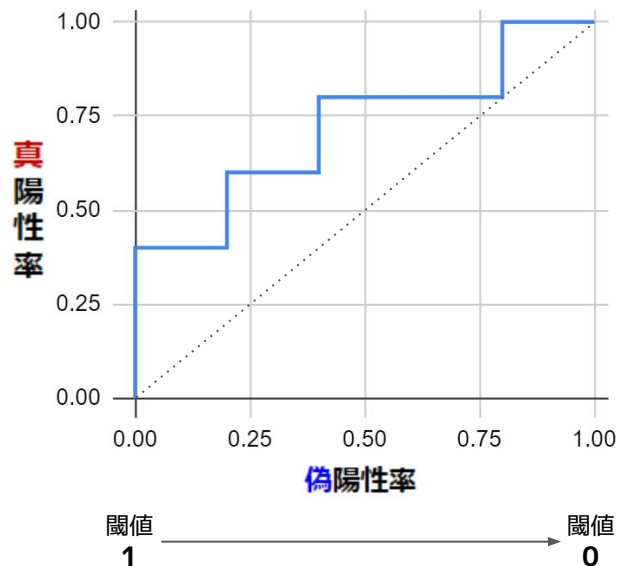




## 概 要

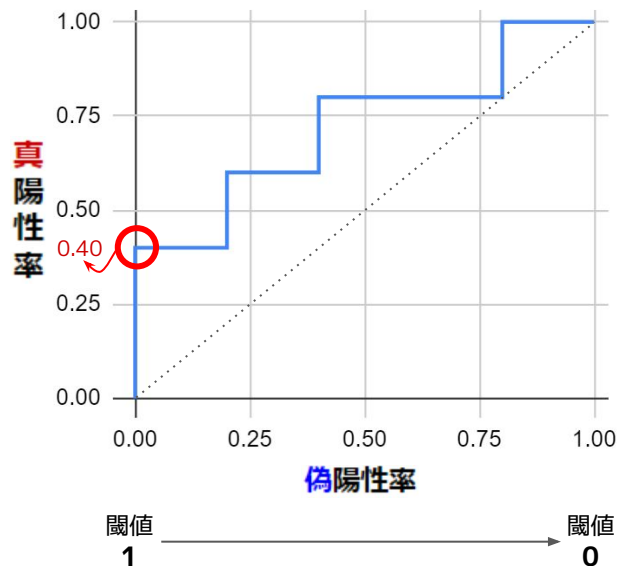
- **ROC曲線**：予測確率の閾値別の真陽性率と偽陽性率をプロットした曲線
- 適切なモデルではこの曲線の下側のエリア (AUC) が広くなる
- AUCの面積比率をスコア化したものが「**ROC-AUCスコア**」
- 比率なので0.0～1.0の間
- **0.5 より上なら当てずっぽうより良い予測**

# ROC曲線の細かい描き方 [補足的トピック]



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

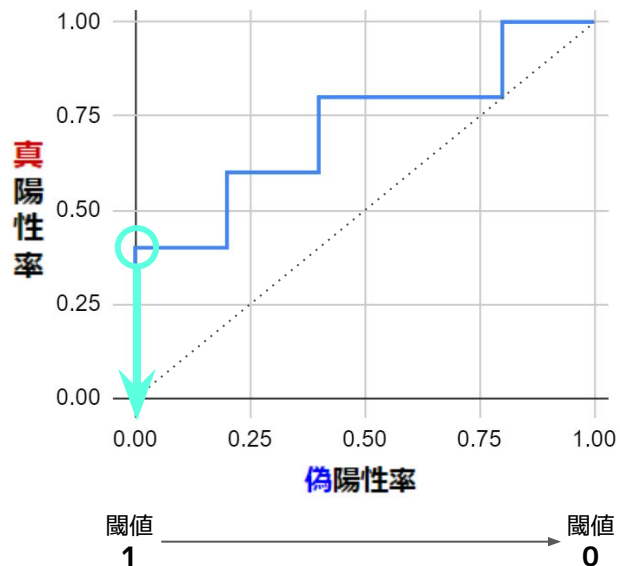
# ROC曲線の細かい描き方 [補足的トピック]



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

正解が陽性のレコード 5 件のうち、正解は 2 件  
よって、真陽性率は  $2 \div 5$  で 0.4 (40%)

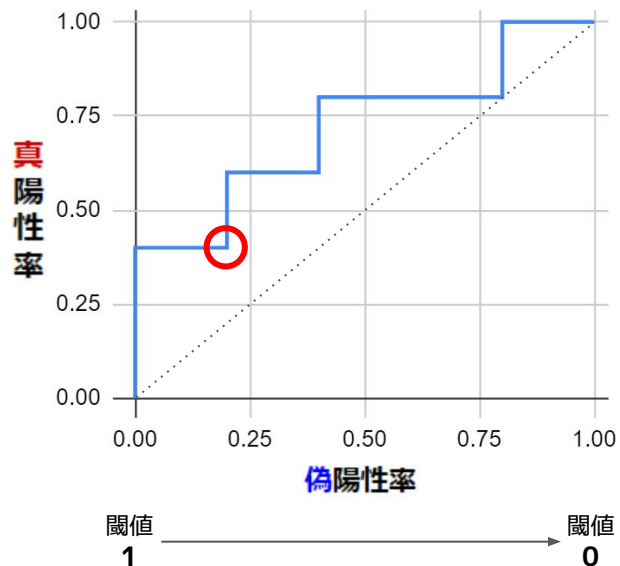
# ROC曲線の細かい描き方 [補足的トピック]



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
正解 003	0	0.83	0	1	1	1	1
004	1	0.75	0	0	1	1	1
正解 005	0	0.61	0	0	1	1	1
006	1	0.52	0	0	1	1	1
正解 007	0	0.44	0	0	0	1	1
正解 008	0	0.35	0	0	0	1	1
009	1	0.29	0	0	0	1	1
正解 010	0	0.11	0	0	0	0	1

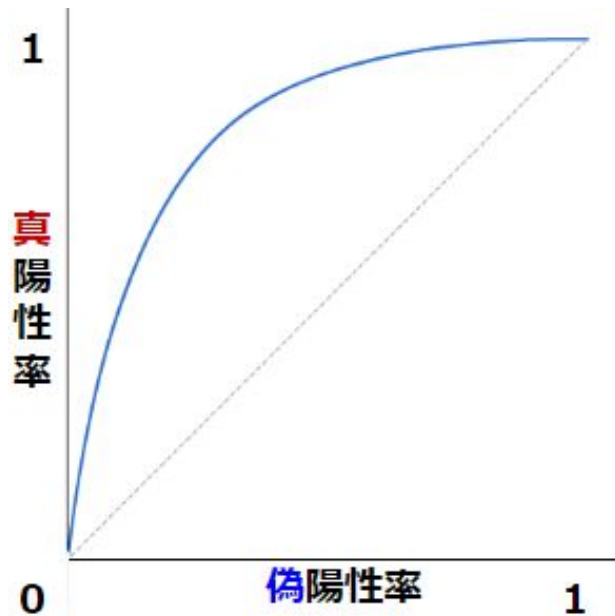
正解が陰性のレコード5件のうち、不正解は0件  
よって、偽陰性率は0%

# ROC曲線の細かい描き方 [補足的トピック]



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

# ROC曲線の細かい描き方 [補足的トピック]



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

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

## 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)[:,-1]

roc_auc_score(正解値y_test, 予測値y_pred)

>> 0.865544
```

# 分類モデルの評価指標：使い分けの観点からまとめ



## Recall

- 空振りよりも**見逃し**の方が顕著にハイリスクな用途のとき  
(がん検診、マルウェア検出、災害予知、児童虐待検出など)

## Precision

- 見逃しよりも**空振り**の方が顕著にハイリスクな用途のとき  
(迷惑メール検出、不可逆的施術の判断、投機的金融商品の買付判断など)

## F-Value(F1)

- **RecallとPrecisionの総合評価**でモデル比較したいとき

## ROC-AUC

- **不均衡データ**でモデル比較をしたい/全ての閾値で総合評価したいとき
- **異なる問題設定間でのモデル比較**をしたいとき

## Accuracy

- 均衡データにおいて**とりあえずお手軽に & 直感的に**精度把握したいとき
- 複雑な評価指標について知らない人に説明するとき

## logloss

- 確率値で出力した予測結果を使って**正解率より細かく**評価を行いたいとき
- (モデルの学習アルゴリズムの内部処理でよく使われている)



# 分類モデルの評価指標：使い分けの観点からまとめ



Recall	<ul style="list-style-type: none"><li>空振りよりも<b>見逃し</b>の方が顕著にハイリスクな用途のとき (がん検診、マルウェア検出、災害予知、児童虐待検出など)</li></ul>
Precision	<ul style="list-style-type: none"><li>見逃しよりも<b>空振り</b>の方が顕著にハイリスクな用途のとき (迷惑メール検出、不可逆的施術の判断、投機的金融商品の買付判断など)</li></ul>
F-Value(F1)	<ul style="list-style-type: none"><li>RecallとPrecisionの<b>総合評価</b>でモデル比較したいとき</li></ul>
ROC-AUC	<ul style="list-style-type: none"><li><b>不均衡データ</b>でモデル比較をしたい/全ての閾値で総合評価したいとき</li><li><b>異なる問題設定間でのモデル比較</b>をしたいとき</li></ul>
Accuracy	<ul style="list-style-type: none"><li>均衡データにおいて<b>とりあえずお手軽に &amp; 直感的に</b>精度把握したいとき</li><li>複雑な評価指標について知らない人に説明するとき</li></ul>
logloss	<ul style="list-style-type: none"><li>確率値で出力した予測結果を使って<b>正解率より細かく</b>評価を行いたいとき</li><li>(モデルの学習アルゴリズムの内部処理でよく使われている)</li></ul>

3

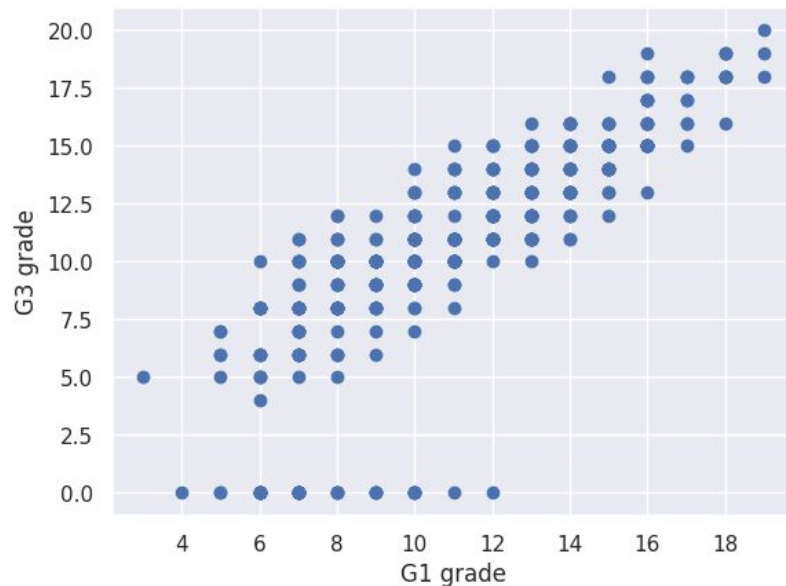
モデルを検証するための評価指標

(a) 分類モデルの評価指標

(b) 回帰モデルの評価指標

- 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らないので、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

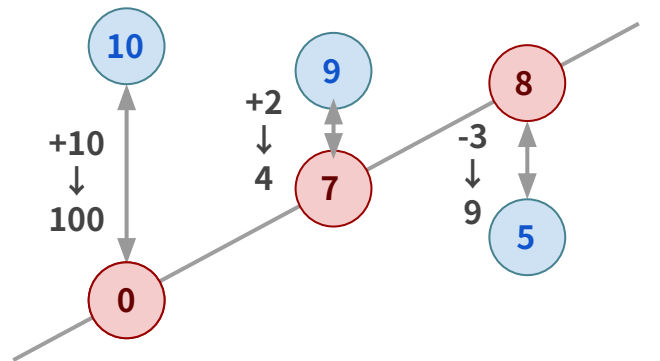
第5回で1学期と3学期の成績をプロットした例



## 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



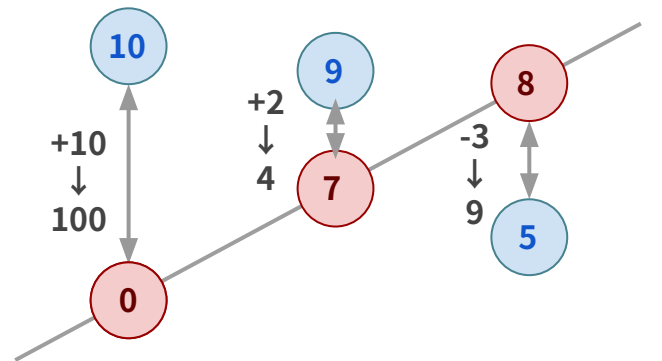
$$MSE = \frac{1}{3} \cdot (100 + 4 + 9) = 37.7$$

● → 予測値  
● → 正解値  
↕ ← 正解値と予測値の誤差

## 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



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

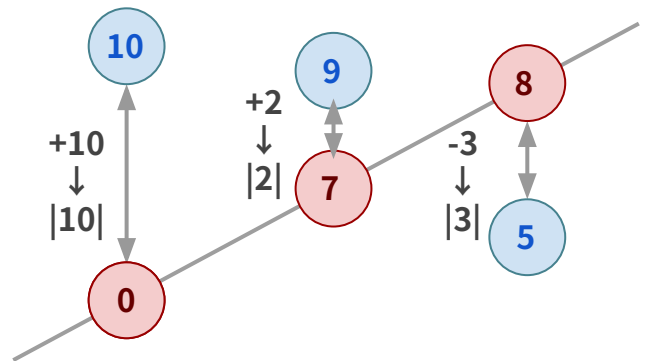
● → 予測値  
● → 正解値

↕ ← 正解値と予測値の誤差

## 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



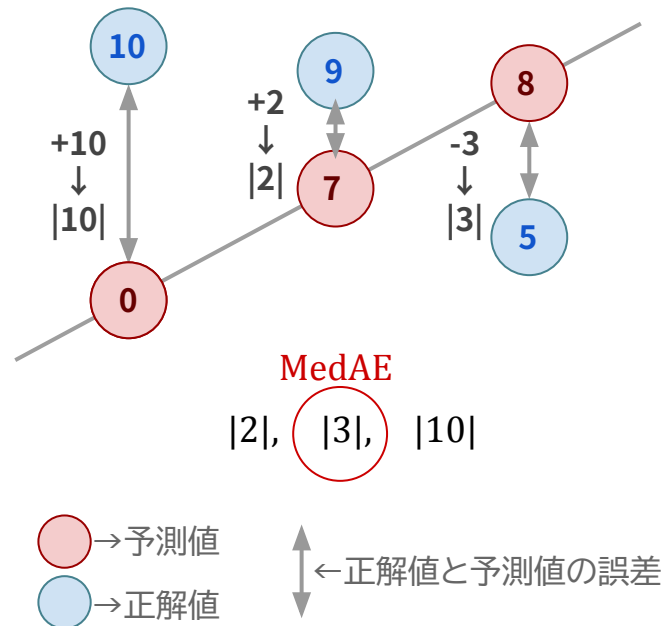
$$MAE = \frac{1}{3} \cdot (|10| + |2| + |3|) = |5|$$

● → 予測値  
● → 正解値  
↕ ← 正解値と予測値の誤差

## MedAE : Median Absolute Error

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

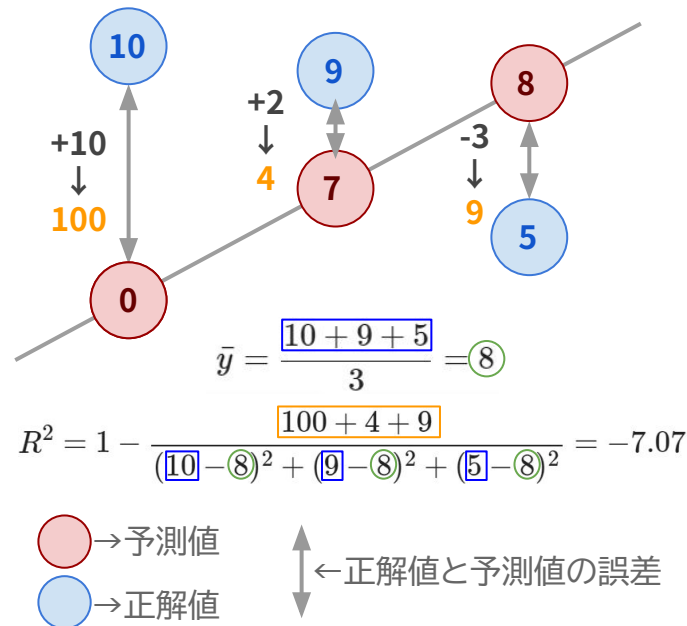
- 平均の代わりに中央値を使って絶対誤差を算出する評価指標
- 誤差平均が歪むレベルの外れ値があっても影響を受けない
- 頻繁にみかける指標ではない



## R2 : R squared

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

- 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に、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は比べて誤差を減らせているかをみるもの





# 回帰モデルの評価指標：使い分けの観点からまとめ



	基本性質	用 途	
Mean Squared Error <b>MSE</b> (平均二乗誤差)	平均的に良い予測か否かより <b>大外しをしない</b> モデルを高く評価	予測誤差が一定以上になると損失が格段に大きくなる用途 (需要予測や投資予測など)	値が小さいほど良い
Mean Absolute Error <b>MAE</b> (平均絶対誤差)	大外しをするか否かより <b>平均的に良い予測をする</b> モデルを高く評価	平均的に正確な予測ができれば大外しが混ざっていても損失を相殺できる用途	
Median Absolute Error <b>MedAE</b> (中央絶対誤差)	MAEよりも更に大胆に大外しを無視してモデルを評価	MSEやMAEと併用してモデルを多角的に評価したいとき	
R squared <b>R2</b> (決定係数)	<b>完全な予測なら1.0、当てずっぽう並みなら0.0</b> 、それより酷ければ負の数が出される	異なる問題設定間でのモデル性能比較をしたいとき	値が1に近いほど良い

# 回帰モデルの評価指標：使い分けの観点からまとめ



	基本性質	用 途	
Mean Squared Error <b>MSE</b> (平均二乗誤差)	平均的に良い予測か否かより <b>大外しをしない</b> モデルを高く評価	予測誤差が一定以上になると損失が格段に大きくなる用途 (需要予測や投資予測など)	値が小さいほど良い
Mean Absolute Error <b>MAE</b> (平均絶対誤差)	大外しをするか否かより <b>平均的に良い予測をする</b> モデルを高く評価	平均的に正確な予測ができれば大外しが混ざっていても損失を相殺できる用途	
Median Absolute Error <b>MedAE</b> (中央絶対誤差)	MAEよりも更に大胆に大外しを無視してモデルを評価	MSEやMAEと併用してモデルを多角的に評価したいとき	
R squared <b>R2</b> (決定係数)	<b>完全な予測なら1.0、当てずっぽう並みなら0.0</b> 、それより酷ければ負の数が出される	異なる問題設定間でのモデル性能比較をしたいとき	値が1に近いほど良い

# 回帰モデルの評価指標：単位の観点からまとめ



<b>MSE</b> (平均二乗誤差)	解釈不可能な相対値 MSEの平方根をとれば目的変数と同じ単位になる = <b>平均平方二乗誤差</b> ( <b>RMSE</b> : Root Mean Squared Error)	値が小さいほど良い
<b>MAE</b> (平均絶対誤差)	目的変数と同じ単位  (MAEが10なら、そのモデルの予測値は正解値の概ね±10の範囲内に収まると解釈できる)	
<b>MedAE</b> (中央絶対誤差)		
<b>R2</b> (決定係数)	相対値 (目的変数とは異なる)	値が1に近いほど良い

## 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

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

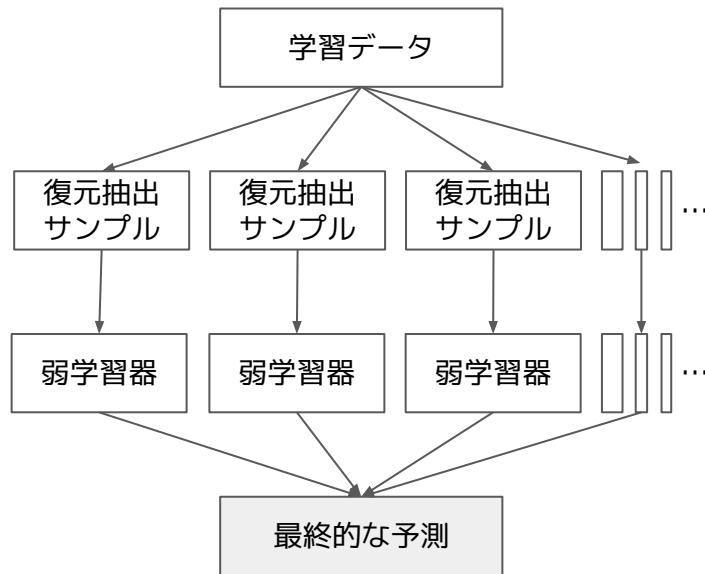
モデルを検証するための評価指標

4

汎化性能向上のためのアンサンブル手法

## 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



## 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
```

## 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>

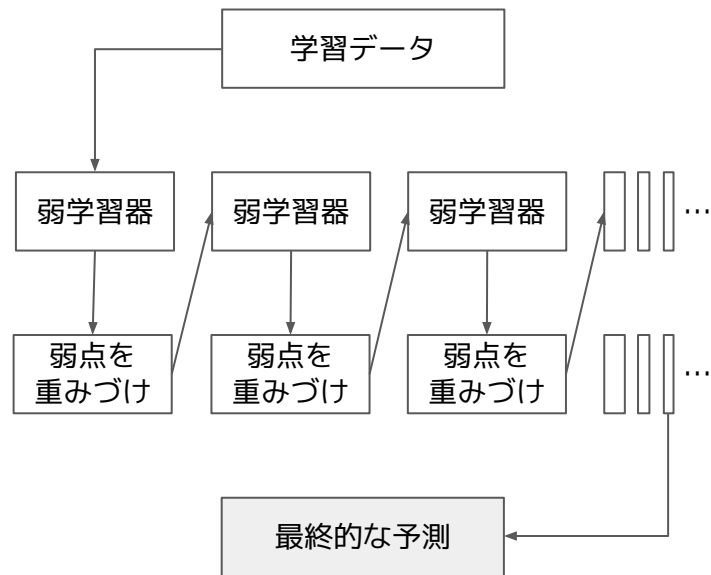
## 主なメソッド

- **.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



## 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



## 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
```

## 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。

## 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,
    eval_metric='rmse',
    early_stopping_rounds=20
)

reg.fit(X_train, y_train,
        eval_set=[(X_valid, y_valid)])
y_pred = reg.predict(X_test) # 予測
```

## 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
```

# アンサンブル：使い分けの観点からまとめ



バギング	RandomForest	過学習を避けたいとき 有用な特徴量の特定ができていないプロジェクト初期段階でのベースラインモデルとして	バギングとしては第一選択肢
	BaggingClassifier /Regressor		決定木系以外のモデルが適する用途で
ブースティング	<b>XGBoost</b> GradientBoosting	精度追求をしたいとき ある程度EDAで有用な特徴量が見えてきた段階など	多くの場合で第一選択肢になる
	AdaBoost		決定木系以外のモデルが適する用途で
アンサンブル 不使用	単体モデル	結果の解釈性が重要なとき	

# 今日の授業では最小限、この知識が頭に残っていればOK



- **クロスバリデーション**とはデータセットを  $k$  個に分割し、各回の検証結果で性能評価すること
- **グリッドサーチ**は総当たりで、**ランダムサーチ**は無作為抽出でハイパラ探索する
- 分類タスクで**再現率**が高ければ見逃しが少なく、**適合率**が高ければ空振りが少ない
- 分類タスクで**ROC-AUC**が0.5を超えていれば当てずっぽうより良い予測ということ
- 回帰タスクで予測の大外しを厳しく評価したければ**MSE**、そうでなければ**MAE**を使う
- 回帰タスクで**R2スコア**が0を超えていれば当てずっぽうより良い予測ということ
- 複数モデルの予測を組み合わせるアンサンブルにおいて、  
**バギング**は過学習対策に優れ、**ブースティング**は学習不足対策や精度追求に優れる
- 上記いずれも専用の関数を使えば自動のできるので**細かい話は暗記しなくても使える**



# 話が多かったと感じた人はまずはこの関数が使えればOK

## クロスバリデーション

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

分類モデルの予測値を確率値で出す

```
y_pred = model.predict_proba(X_test)[: ,1]
```

## 勾配ブースティングツリー

```
                回帰の場合          分類の場合  
model = XGBRegressor() / XGBClassifier()  
  
model.fit(X_train, y_train) # 学習  
  
y_pred = model.predict(X_test) # 予測
```

## 評価指標系

### ROC-AUCスコア

```
roc_auc_score(y_test, y_pred)
```

### 再現率

```
recall_score(y_test, y_pred)
```

### 適合率

```
precision_score(y_test, y_pred)
```

### 決定係数 (R2スコア)

```
r2_score(y_test, y_pred)
```

### 平均二乗誤差 (MSE)

```
mean_squared_error(y_test, y_pred)
```

「GCIが終わった後」、更にその先を目指す際におすすめの学習項目（GCI中は忘れてOK）

- **Git & Github**（『[実務レベルでわかる／使いこなせるようになる Git入門コマンドライン演習80](#)』）
- **Docker**（『[仕組みと使い方がわかる Docker & Kubernetesのきほんのきほん](#)』）
- **IDE (統合開発環境)**（[Visual Studio Code](#) など、好みに合うものを適宜調べる）
- **オブジェクト指向プログラミング & クリーンアーキテクチャリング**
  - **GoogleColab & JupyterNotebookから離れた開発ができるスキル**（.pyファイルに処理をまとめてコマンドラインで実行できるようにするために必要な知識など）
  - 『[Clean Architecture 達人に学ぶソフトウェアの構造と設計](#)』（※名著ですが中級以上向けの本ではありません）  
※ この本が難しい方はまずはこの記事（[Qiita記事：データサイエンティストのためのデータクラス](#)）などをご一読いただくことからでも良いかと思います
  - 「[Pythonコーディング規約 PEP8](#)」（[解説サイトの例](#)）
- **クラウドサーバ**
  - AWS, GCP, Azure（沼なので一つ軽く試してみる程度でも十分／入り口はGCPの[Cloud Run](#)あたりがおすすめ）
- **MLOpsツール**
  - [Weights & Biases](#), [mlflow](#), [Airflow](#)など（沼なので一つ軽く試してみる程度でも十分／Weights & Biasesがおすすめ）
- **UNIX互換OSでのコマンドライン操作の初歩的な知識**（沼なのでちょっとで大丈夫）

- データサイエンスコンペティション
  - [SIGNATE](#)
    - 「Beginner限定コンペ」が最初の挑戦におすすめ
  - [Nishika](#)
    - 上位入賞すると企業の採用担当者から声がかかる
  - [Kaggle](#)
    - 最もメジャーだが難易度は高め
    - 「称号」を得るとかなりオーセンティックな経歴になる
- 参考書籍
  - 『[Kaggleで勝つデータ分析の技術](#)』