リサンプリング法

概要

- リサンプリング法（resampling methods）
  - 手持ちのデータからのランダムサンプリング＆学習（パラメータ推定）を複数回行い、モデルに関する様々な情報を得る
  - 計算コストは高いが計算機の能力向上によりコストの問題は小さくなっている

- 代表的なリサンプリング手法
  - 交差検証（cross-validation）
    - モデル評価（model assessment）
    - モデル選択（model selection）
  - ブートストラップ（bootstrap）
    - パラメータの信頼性の評価

性能評価

- 学習したモデルの性能評価
  - 回帰問題
    - 平均二乗誤差（mean squared error, MSE）, etc.
  - 分類問題:
    - エラー率（error rate）, 感度（sensitivity）, etc.

- 未知のデータに対する性能が知りたい
  - 学習データでの誤差を小さくすることは簡単
    - 予測変数を増やす、KNN で $K = 1$ にする、等
  - 学習誤差は性能の指標としてはあまり意味がない

→ 手持ちのデータの一部を性能評価用にとっておく

検証セットによる手法

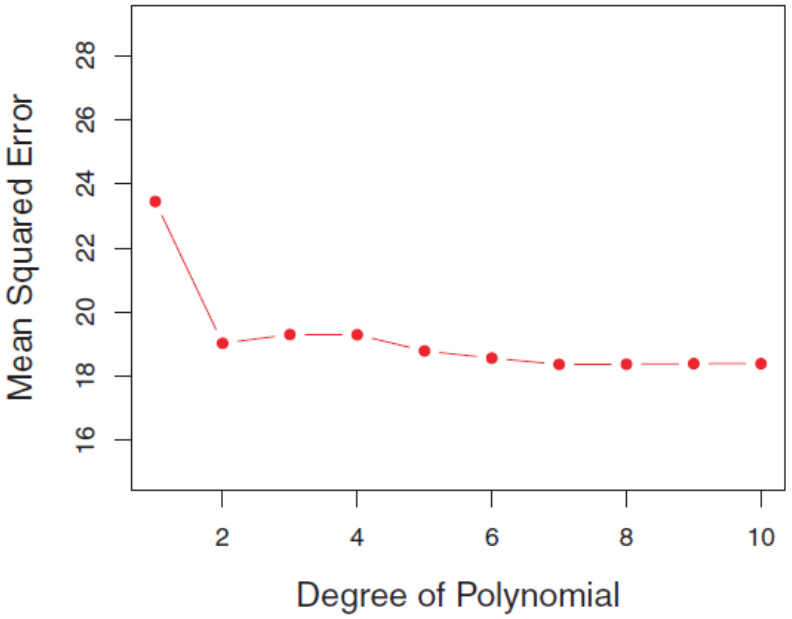
- 検証セット（validation set）による手法
  - データセットを（ランダムに）2つに分割
    - 学習用データセット（training set）
    - 検証用データセット（validation set, hold-out set）

  - 学習用データでモデルのパラメータ等を推定し、検証用データで精度を評価する

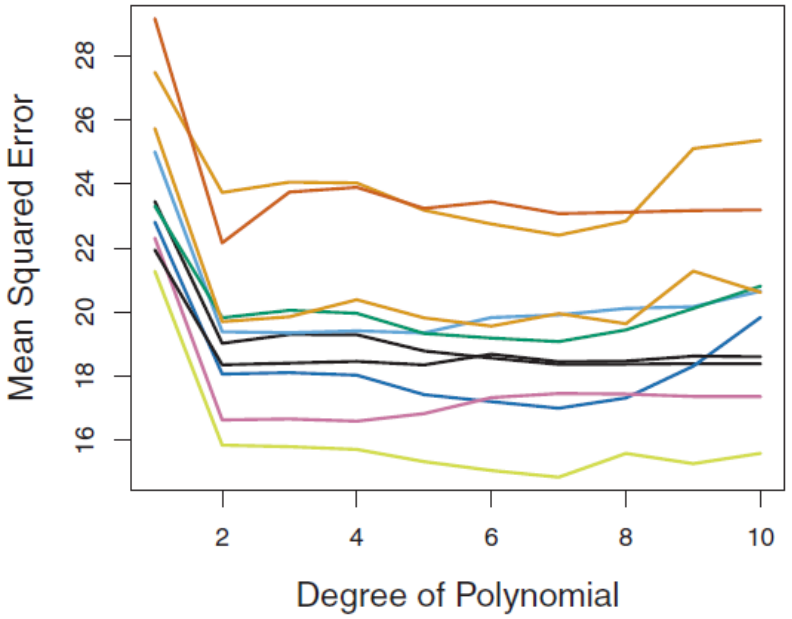
検証セットによる手法

- Autoデータセット
  - mpg を horsepower の多項式関数で予測

392個の事例の半分を検証用に使用



異なる分割による結果（10回分）



検証セットによる手法

- 問題点
  - データがどう分割されたかによって精度の推定値が大きく変わる
  - モデルの学習（パラメータ推定）にはデータ全体の半分しか使われない
    - 学習に使えるデータが少ないため、精度が本来達成可能な精度よりも低めに推定される

交差検証

- Leave-One-Out 交差検証（Leave-One-Out Cross-Validation, LOOCV）
  - ひとつの事例を除いたすべてのデータを学習データとして利用し、モデルを学習
  - 学習に利用されなかった事例で性能評価
  - 上記手順をすべての事例に対して行い、評価指標の平均を最終的な評価指標の推定値とする

交差検証

- Leave-One-Out (LOO) Cross-Validation（LOO交差検証）

全データ $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\}$

学習データ $\{(x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\}$ ➡ $\mathrm{MSE}_1 = (\hat{y}_1 - y_1)^2$

学習データ $\{(x_1, y_1), (x_3, y_3), \ldots, (x_n, y_n)\}$ ➡ $\mathrm{MSE}_2 = (\hat{y}_2 - y_2)^2$

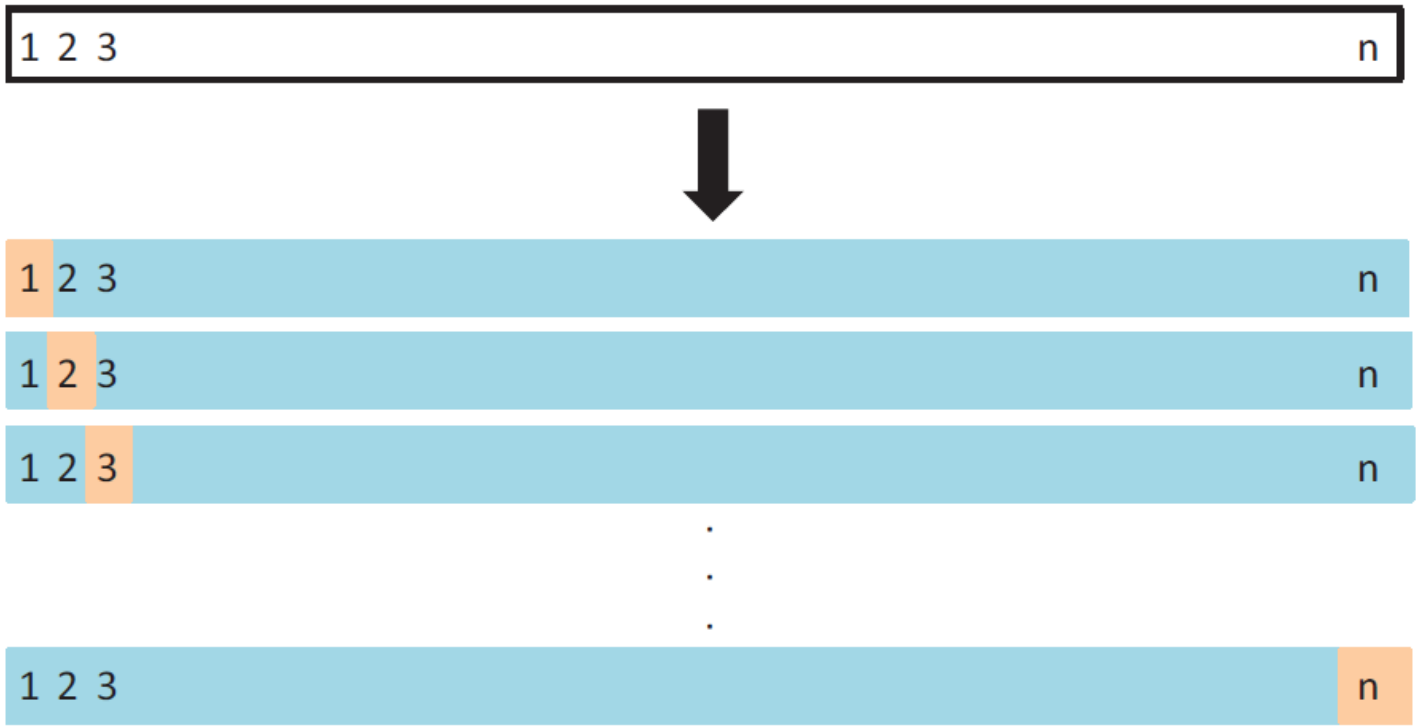
… …

学習データ $\{(x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})\}$ ➡ $\mathrm{MSE}_n = (\hat{y}_n - y_n)^2$

$$\mathrm{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n} \mathrm{MSE}_i$$

交差検証

- Leave-One-Out (LOO) Cross-Validation（LOO交差検証）



交差検証

- Leave-One-Out 交差検証の利点
  - 学習には *n*-1 個の事例が使われる
    - データ全体を使った場合とほとんど同じ性能のモデルが作れるので、性能が低く見積もられることがない
  - 分割の違いによる性能評価のばらつきがない

- Leave-One-Out 交差検証の欠点
  - *n*回の学習と評価を行うので計算コストが高い
    - モデルによっては効率的に計算することができる場合もある

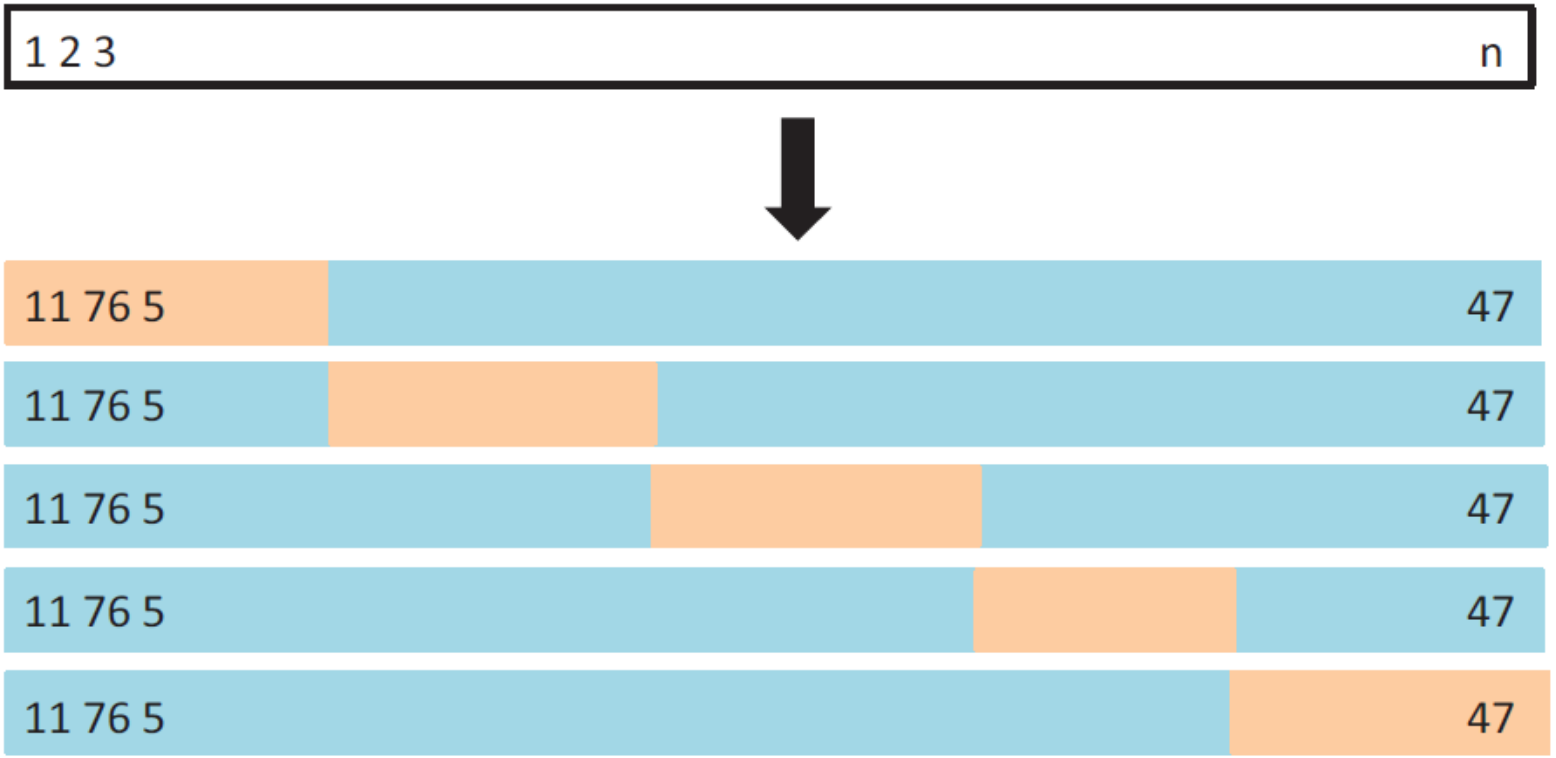
k-分割交差検証

- *k*-分割交差検証（*k*-fold cross-validation）
  - データを *k* 個のグループに分割
  - ひとつを検証データ、残りの *k* – 1 個を学習データとして MSE を計算
  - 検証データに使うグループを変えて *k* 回繰り返す
  - テスト MSE は *k* 個の MSE の平均として推定

$$\mathrm{CV}_{(k)} = \frac{1}{k}\sum_{i=1}^{k}\mathrm{MSE}_i$$

k-分割交差検証

- 5分割交差検証



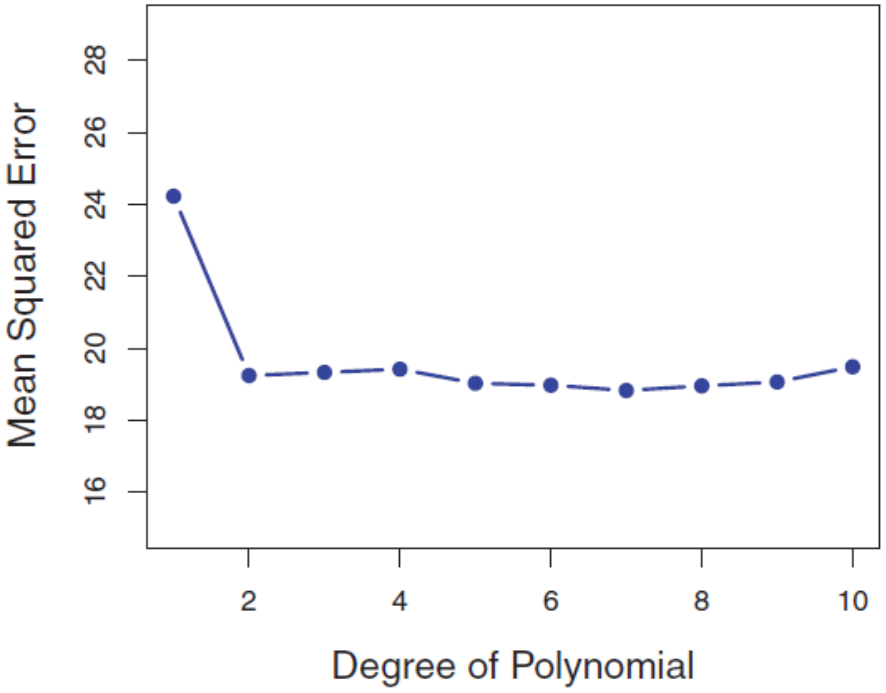
k-分割交差検証

- *k*-分割交差検証 vs LOO交差検証
  - LOO交差検証は一般に *n* 回の学習を行う必要があるのに対して、*k*-分割交差検証の場合は *k* 回で済む
  - LOO交差検証は、*k*-分割交差検証で *k* を *n* とした場合に相当
  - Bias-variance trade-off
    - *k*-分割交差検証は学習に使えるデータが少し少ないのでバイアスが大きい
    - LOO交差検証は、（学習に使うデータが毎回ほとんど同じなので）出力間に強い相関があり、バリアンスが大きい
- *k* = 5 or 10 程度がよく用いられる

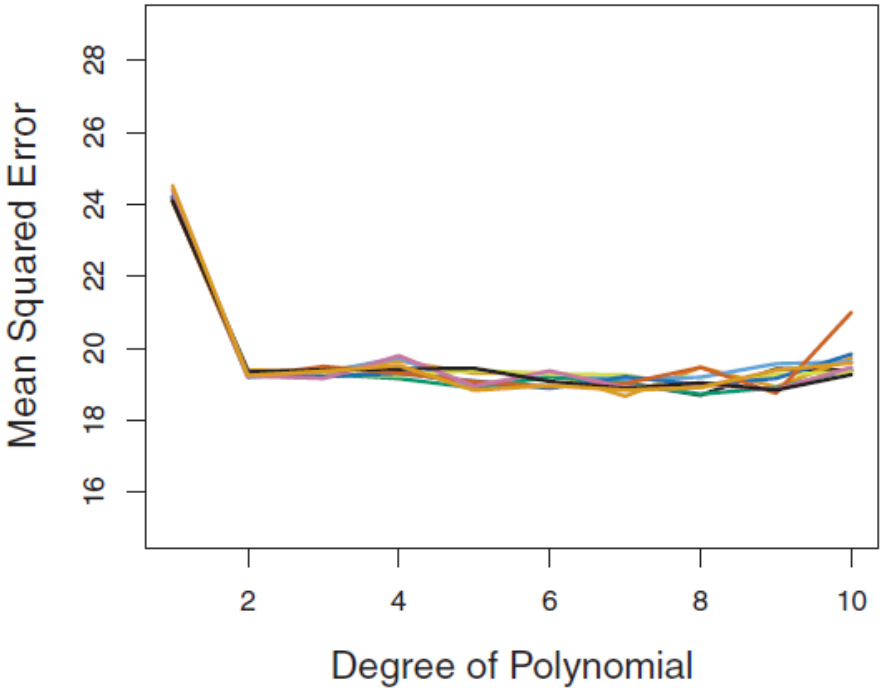
交差検証

- Autoデータセット
  - mpg を horsepower の多項式関数で予測

LOOCV



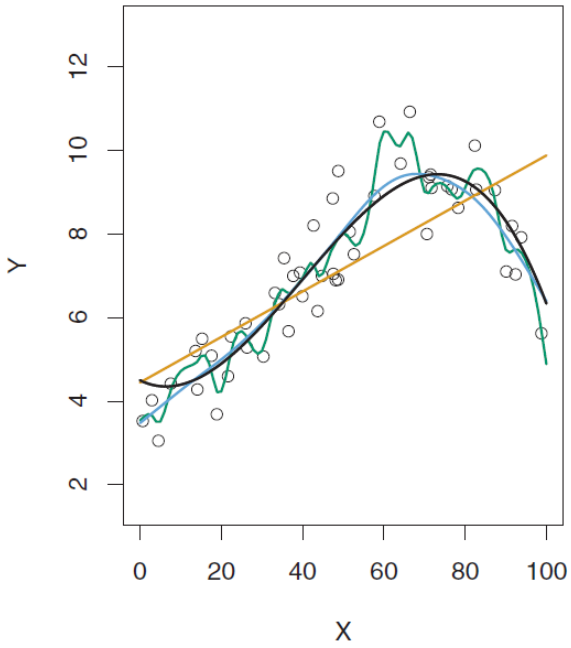
10-fold CV

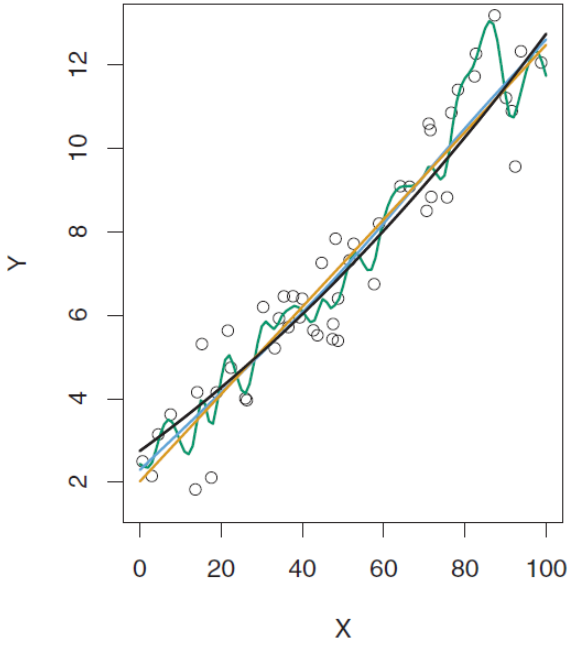


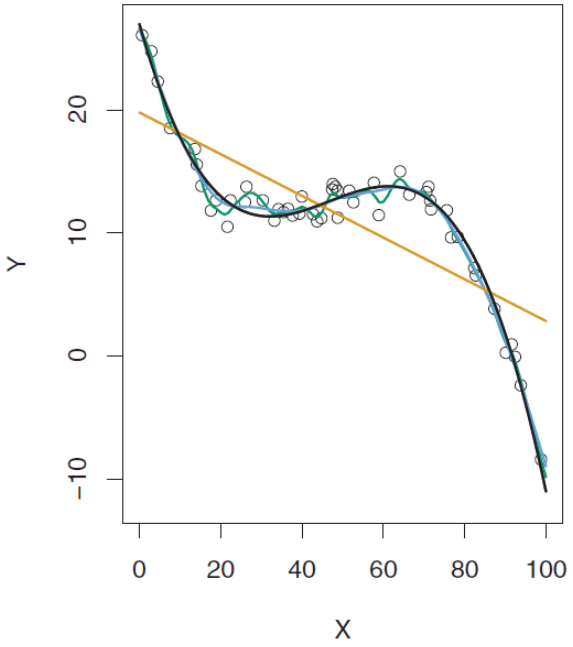
交差検証の精度

- 例）スプライン回帰（再掲）

黒：真の f
オレンジ：線形回帰
青、緑：平滑化スプライン



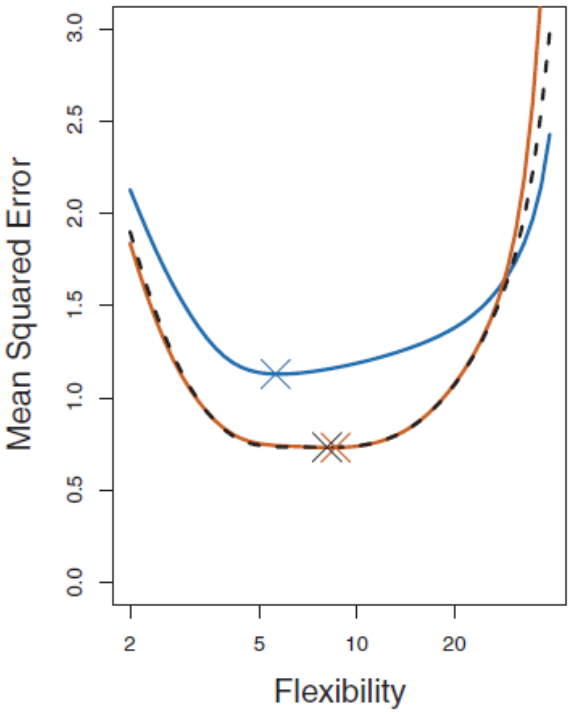


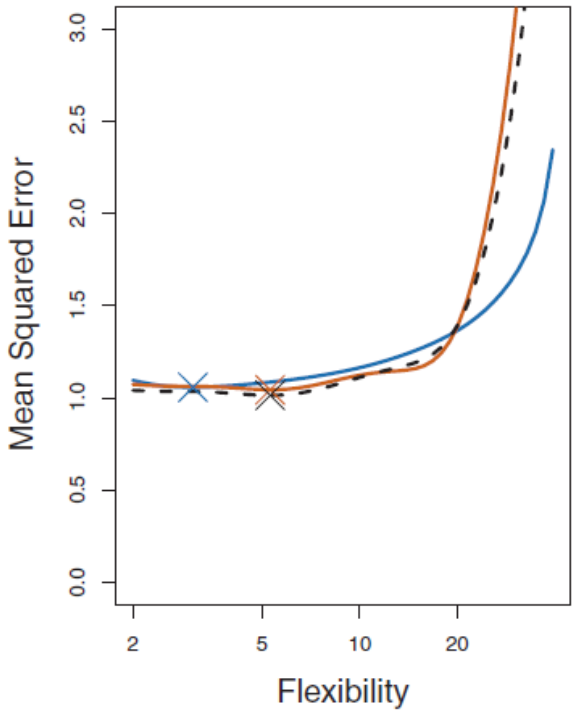


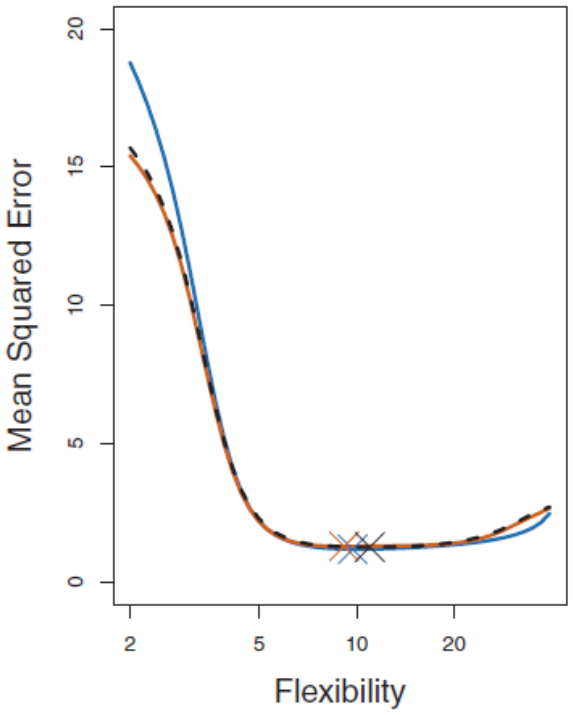
交差検証の精度

- 交差検証によるテストエラーの推定

青線：テストMSE
黒の破線：LOOCV
オレンジ：10-fold CV







交差検証

- 分類問題の場合
  - エラーの定義以外は回帰問題と同様
  - LOO交差検証エラー

$$\mathrm{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\mathrm{Err}_i \qquad \mathrm{Err}_i = I\left(y_i \neq \hat{y}_i\right)$$

  - *k*-分割交差検証も同様
    - MSEの代わりにエラー率を計算

交差検証

紫：ベイズ決定境界
黒：ロジスティック回帰決定境界

- 例
  - ロジスティック回帰

Degree = 1

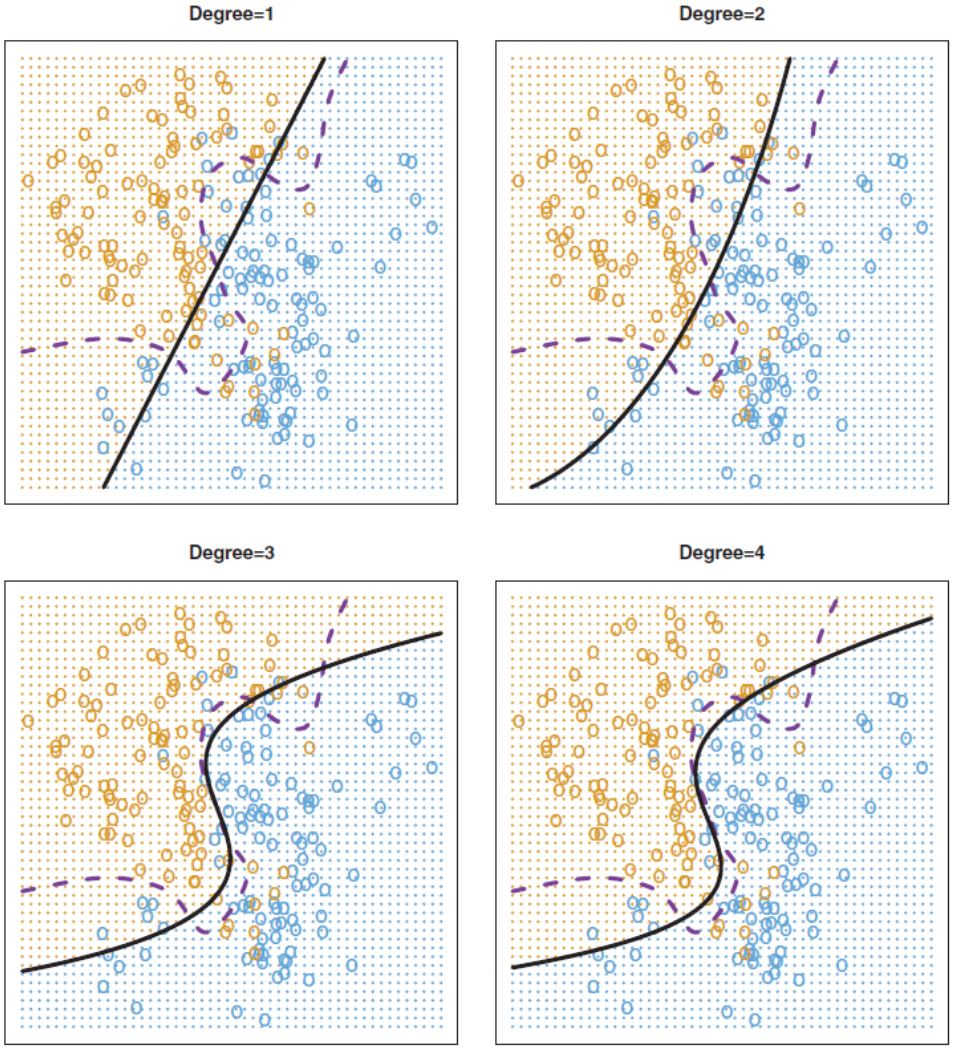
$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$

Degree = 2

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2^2 + \beta_3 X_2 + \beta_4 X_2^2$$

Degree = 3

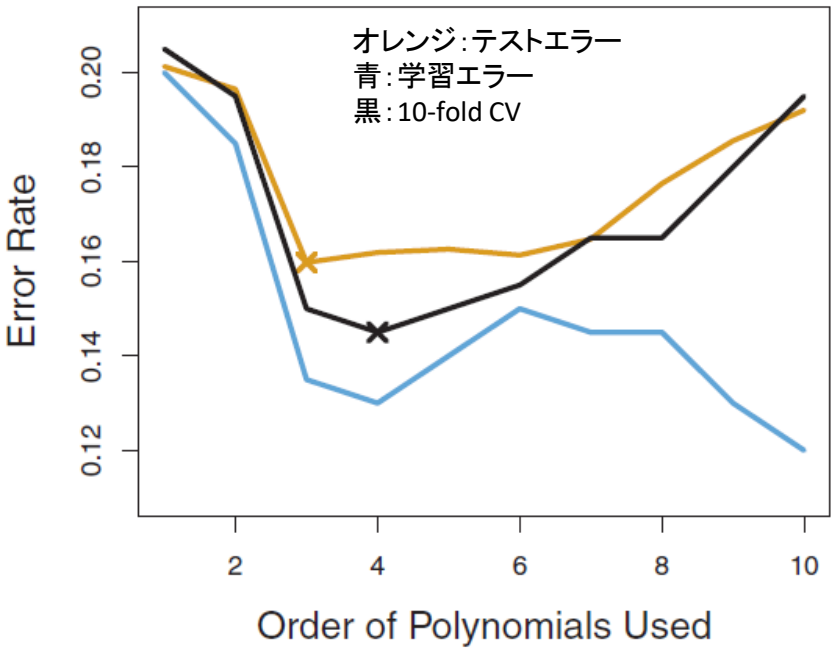
…



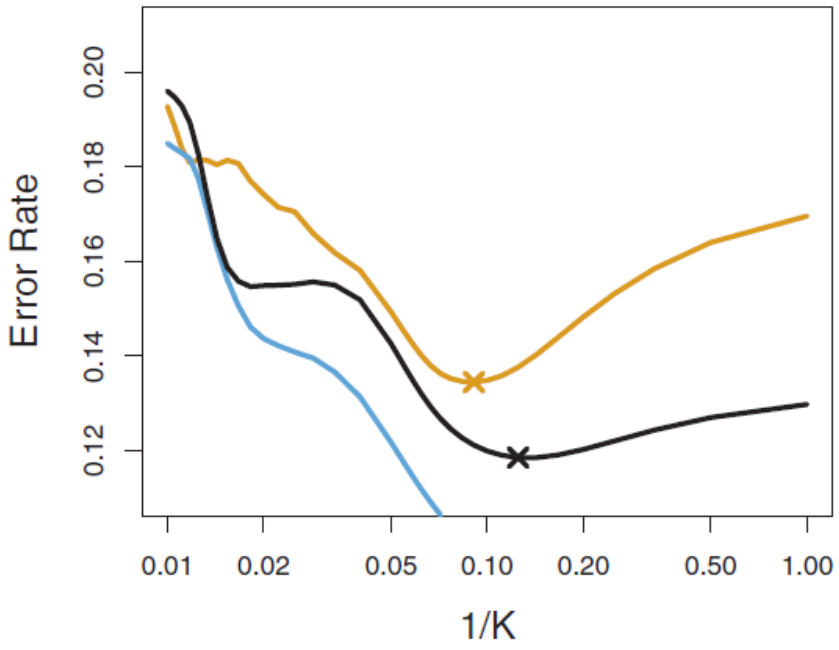
交差検証

- 10分割交差検証によるエラー率の推定

ロジスティック回帰



KNN



Python実習

- 外部データ（csv形式）の読み込み
  - ダウンロード
    - サンプルファイル　Auto.csv
      - http://www-bcf.usc.edu/~gareth/ISL/Auto.csv
    - 上記ファイルをホームディレクトリに保存（Windowsの場合）
  - ホームディレクトリに移動

```
>>> import os
>>> os.chdir(os.path.expanduser("~"))
```

  - `pandas.read_csv()`関数でデータフレームとして読み込み

```
>>> import pandas as pd
>>> Auto = pd.read_csv('Auto.csv', header=0, na_values='?')
>>> Auto
...
```

Python実習

- 検証セットによる精度評価
  - データの分割

```
>>> Auto = Auto.dropna().reset_index(drop=True)
>>> Auto.shape
>>> import numpy as np
>>> np.random.seed(1)
>>> train = np.random.choice(len(Auto), 196, replace=False)
>>> train
>>> select = np.in1d(range(len(Auto)), train)
>>> select
```

  - 学習

```
>>> import statsmodels.formula.api as smf
>>> lm = smf.ols('mpg ~ horsepower', data=Auto[select]).fit()
>>> lm.summary()
```

Python実習

- 精度評価
  - 評価

```
>>> preds = lm.predict(Auto)
>>> squared_error = (Auto['mpg'] - preds)**2
>>> np.mean(squared_error[select])
24.62301015144335          ← 学習セットでのエラー
>>> np.mean(squared_error[~select])
23.361902892587246         ← 検証セットでのエラー
```

Python実習

- 検証セットによる精度評価
  - 次数の大きいモデルではどうか？

```
>>> lm2 = smf.ols('mpg ~ horsepower + I(horsepower ** 2.0)',
data = Auto[select]).fit()
>>> preds = lm2.predict(Auto)
>>> squared_error = (Auto['mpg'] - preds)**2
>>> np.mean(squared_error[select])
...
>>> np.mean(squared_error[~select])
...
```

  - 乱数のseedを変えたらどうなるか？

```
>>> np.random.seed(2)
>>> train = np.random.choice(Auto.shape[0], 196, replace=False)
>>> select = np.in1d(range(Auto.shape[0]), train)
...
```

Python実習

- Leave-One-Out 交差検証
  - scikit-learn を利用する

```
>>> x = pd.DataFrame(Auto.horsepower)
>>> y = Auto.mpg
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression()
>>> model.fit(x, y)
>>> model.intercept_
>>> model.coef_
```

Python実習

- Leave-One-Out 交差検証
  - KFold, cross_val_score
  - n_splits を要素数にすれば LOO になる

```
>>> from sklearn.model_selection import KFold, cross_val_score
>>> k_fold = KFold(n_splits=len(x))
>>> test = cross_val_score(model, x, y, cv=k_fold,  scoring =
'neg_mean_squared_error', n_jobs=-1)
>>> np.mean(-test)
...
```

Python実習

- Leave-One-Out 交差検証
  - for ループを用いて異なる次数での精度を調べる

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> from sklearn.pipeline import Pipeline
>>> MSE = []
>>> for porder in range(1, 6):
>>>     model = Pipeline([('poly', PolynomialFeatures(degree=porder)),
('linear', LinearRegression())])
>>>     k_fold = KFold(n_splits=len(x))
>>>     test = cross_val_score(model, x, y, cv=k_fold,  scoring =
'neg_mean_squared_error', n_jobs=-1)
>>>     MSE.append(np.mean(-test))
>>> MSE
...
```

Python実習

- *k*-分割交差検証
  - K を指定

```
>>> MSE = []
>>> for porder in range(1, 6):
>>>    model = Pipeline([('poly', PolynomialFeatures(degree=porder)),
('linear', LinearRegression())])
>>>    k_fold = KFold(n_splits=10)
>>>    test = cross_val_score(model, x, y, cv=k_fold,  scoring =
'neg_mean_squared_error', n_jobs=-1)
>>>    MSE.append(np.mean(-test))
>>> MSE
...
```

ブートストラップ

- ブートストラップ（bootstrapping）
  - 標本からの復元抽出によるランダムサンプリングを繰り返すことで行う統計的推論手法
  - 確率モデルのパラメータの信頼区間などを簡単に求めることができる
    - 線形回帰モデルではパラメータの信頼区間が解析的に得られたが、それが困難な複雑なモデルにも適用可能
  - 例
    1. 手持ちのデータセットから復元抽出によるランダムサンプリングによって新たなデータセットを生成
    2. それを学習データとしてモデルのパラメータを推定
    3. 上記の１～２を多数回繰り返す
    4. 得られたパラメータの分布から信頼区間などを計算

ブートストラップ

- 例）金融資産のポートフォリオ
  - *X*: 金融資産 A のリターン（収益）
  - *Y*: 金融資産 B のリターン（収益）
  - 金融資産 A と B をどんな割合で保有すればリスク（分散）

$$\mathrm{Var}\left(\alpha X + (1-\alpha)Y\right)$$

を最小化できるか？

$$\alpha = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}}$$

$$\sigma_X^2 = \mathrm{Var}(X)$$
$$\sigma_Y^2 = \mathrm{Var}(Y)$$
$$\sigma_{XY} = \mathrm{Cov}(X, Y)$$

参考

リスク最小化

- 最適な α の導出

$$\begin{aligned}\mathrm{Var}(\alpha X+(1-\alpha)Y) &= E[(\alpha X+(1-\alpha)Y)^2]-E[(\alpha X+(1-\alpha)Y)]^2\\ &= \alpha^2(E[X^2]-E[X]^2)-(1-\alpha)^2(E[Y^2]-E[Y]^2)\\ &\quad +2\alpha(1-\alpha)(E[XY]-E[X]E[Y])\\ &= \alpha^2\mathrm{Var}(X)-(1-\alpha)^2\mathrm{Var}(Y)+2\alpha(1-\alpha)\mathrm{Cov}(X,Y)\end{aligned}$$

$\frac{d}{d\alpha}\mathrm{Var}(\alpha X+(1-\alpha)Y)=0$ より

$$\alpha=\frac{\mathrm{Var}(Y)-\mathrm{Cov}(X,Y)}{\mathrm{Var}(X)+\mathrm{Var}(Y)-2\mathrm{Cov}(X,Y)}$$

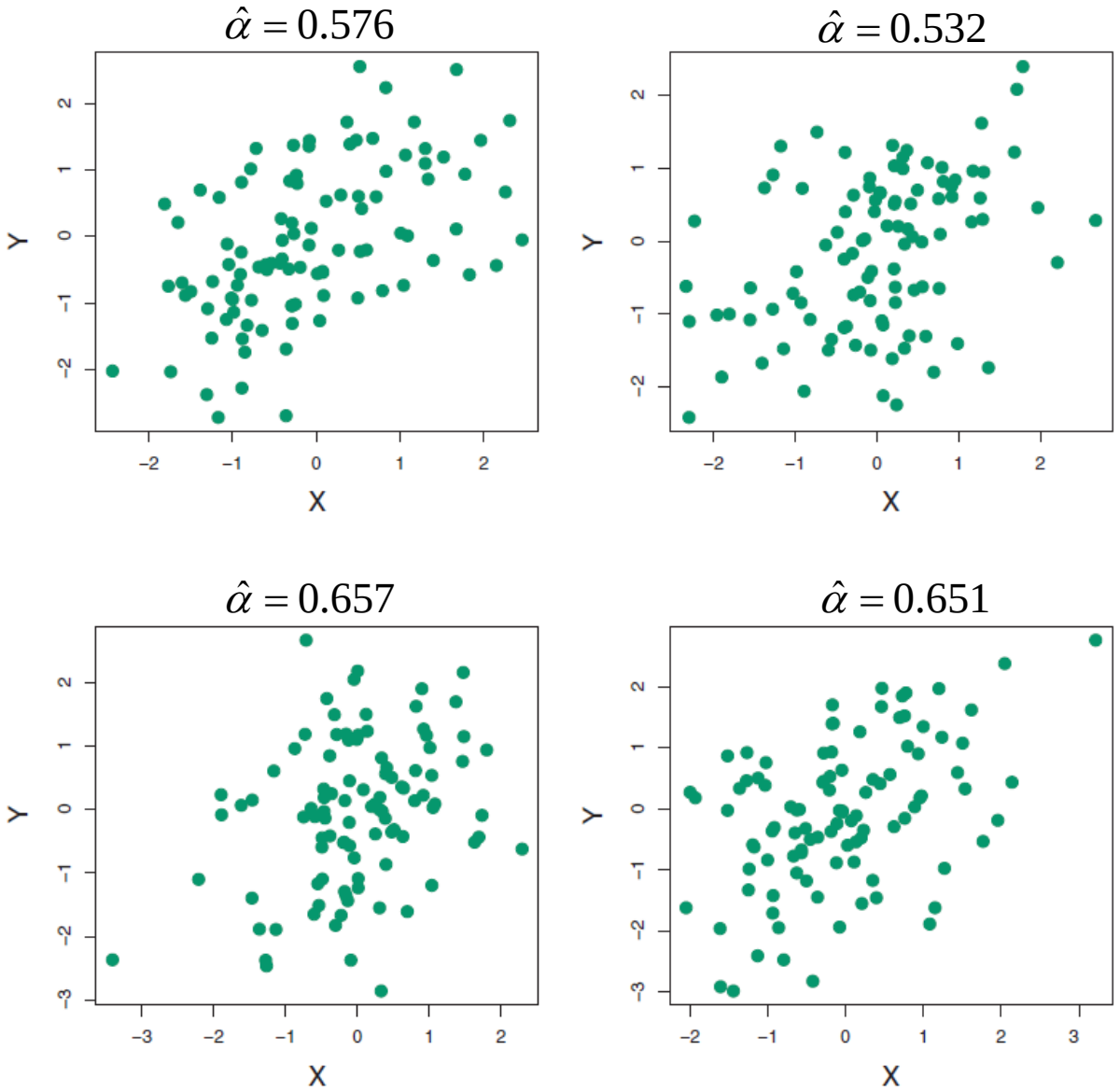
ブートストラップ

- 現実には $\sigma_X^2, \sigma_Y^2, \sigma_{XY}$ は未知なのでデータから推定

- それぞれ100個のデータ点から $\sigma_X^2, \sigma_Y^2, \sigma_{XY}$ を推定し α を推定

$$\hat{\alpha} = \frac{\hat{\sigma}_Y^2 - \hat{\sigma}_{XY}}{\hat{\sigma}_X^2 + \hat{\sigma}_Y^2 - 2\hat{\sigma}_{XY}}$$

$\hat{\alpha}$ の標準偏差は？



ブートストラップ

- 母集団からサンプリングすることができるのであれば

αの推定を1000回行った結果

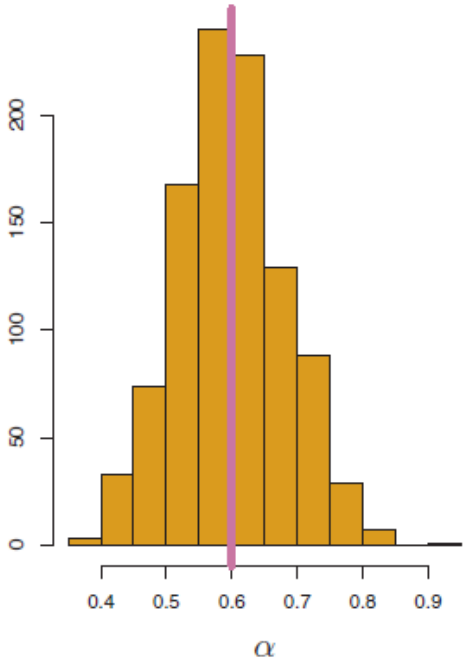
母集団

$\sigma_X^2 = 1$

$\sigma_Y^2 = 1.25$

$\sigma_{XY} = 0.5$

$\alpha = 0.6$



平均

$$\bar{\alpha} = \frac{1}{1000}\sum_{r=1}^{1000}\hat{\alpha}_r = 0.5996$$

標準偏差

$$\sqrt{\frac{1}{1000-1}\sum_{r=1}^{1000}\left(\hat{\alpha}_r - \bar{\alpha}\right)^2} = 0.083$$

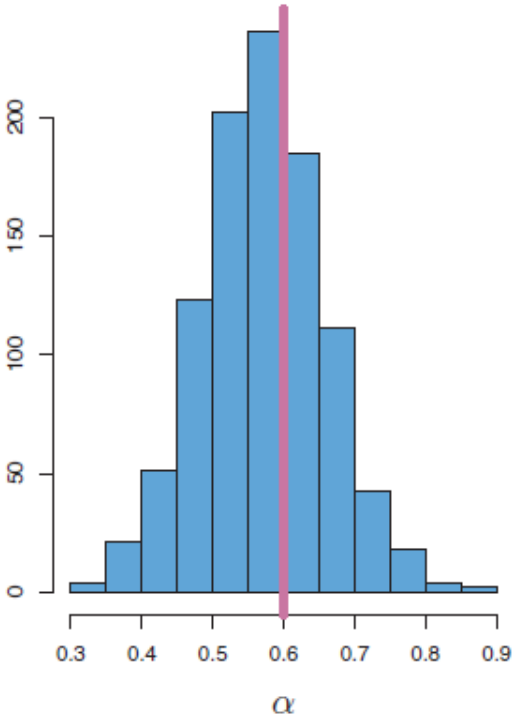
ブートストラップ

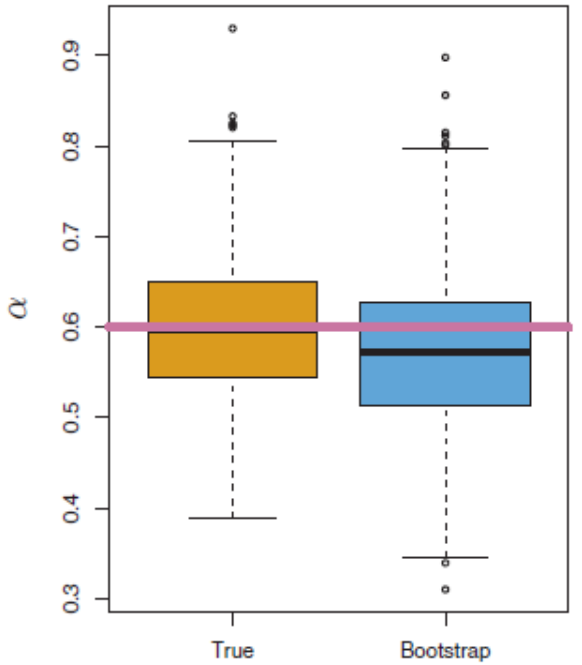
- 実際に手元にあるのは単一のデータセット

100個のデータ点からなる単一のデータセットから
復元抽出で100個のデータ点を抽出しαを計算

1000回繰り返す

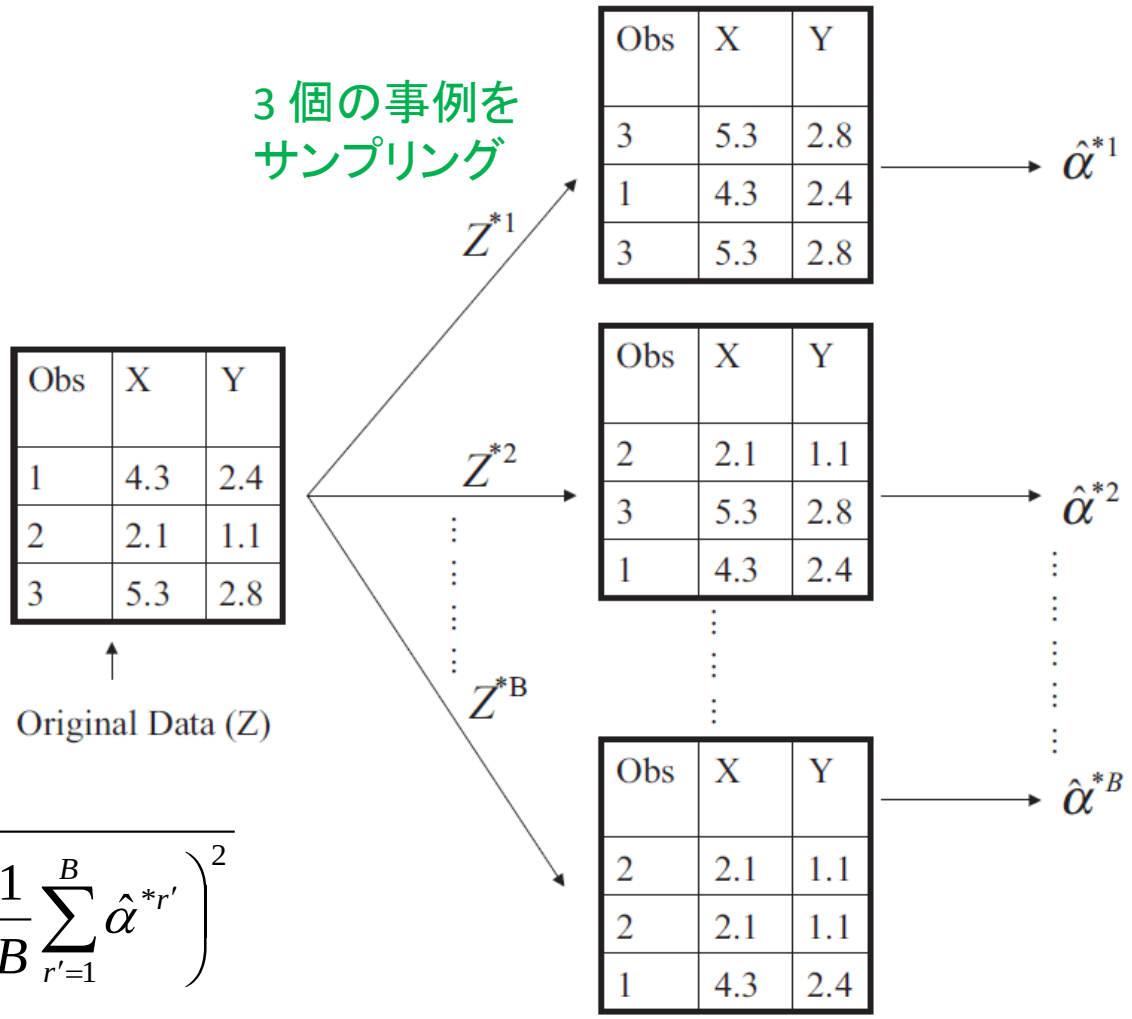
$$\mathrm{SE}_B(\hat{\alpha}) = 0.087$$





ブートストラップ

- 手順



標準誤差の推定

$$\mathrm{SE}_B(\hat{\alpha}) = \sqrt{\frac{1}{B-1}\sum_{r=1}^{B}\left(\hat{\alpha}^{*r} - \frac{1}{B}\sum_{r'=1}^{B}\hat{\alpha}^{*r'}\right)^2}$$

Python実習

- ブートストラップ
  - 例）金融資産のポートフォリオ最適化

```
>>> def alpha_fn(data, index):
>>>     X = data.X[index]
>>>     Y = data.Y[index]
>>>     return (np.var(Y) - np.cov(X,Y)[0,1])/(np.var(X) +
np.var(Y) - 2 * np.cov(X, Y)[0,1])
```

  - Portfolio データセット
    - http://www.logos.t.u-tokyo.ac.jp/~tsuruoka/lecture/sml/Portfolio.csv

```
>>> Portfolio = pd.read_csv('Portfolio.csv', header=0)
>>> alpha_fn(Portfolio, list(range(0, 100)))
...
```

Python実習

- ブートストラップ
  - ランダムサンプリングされたデータでαを推定

```
>>> np.sort(np.random.choice(range(0, 100), size=100,
replace=True))
>>> alpha_fn(Portfolio, np.random.choice(range(0, 100),
size=100, replace=True))
...
>>> alpha_fn(Portfolio, np.random.choice(range(0, 100),
size=100, replace=True))
...
>>> alpha_fn(Portfolio, np.random.choice(range(0, 100),
size=100, replace=True))
...
```

Python実習

- ブートストラップ

```
>>> def bootstrap(data, func, B):
>>>     idx = np.random.randint(0, len(data), (B, len(data)))
>>>     stat = np.zeros(B)
>>>     for i in range(B):
>>>         stat[i] = func(data, idx[i])
>>>     return {'Mean': np.mean(stat), 'STD': np.std(stat)}
>>>
>>> bootstrap(Portfolio, alpha_fn, 1000)
...
>>> bootstrap(Portfolio, alpha_fn, 1000)
...
```

Python実習

- ブートストラップ
  - 線形回帰モデルのパラメータの信頼区間の推定

```
>>> Data = pd.DataFrame({'X': Auto.horsepower, 'Y':Auto.mpg})
>>> def lr_intercept(data, idx):
...     model = LinearRegression()
...     model.fit(pd.DataFrame(data.X[idx]), data.Y[idx])
...     return model.intercept_
>>>
>>> bootstrap(Data, lr_intercept, 1000)
...
```

Python実習

- 参考
  - 線形回帰モデルのパラメータの標準誤差の推定

```
>>> lm.summary()
...
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      40.3338      1.023     39.416      0.000      38.316      42.352
horsepower     -0.1596      0.009    -17.788      0.000      -0.177      -0.142
```

ブートストラップで推定した値と多少異なる

→ ブートストラップの方が仮定が少ないのでより正確