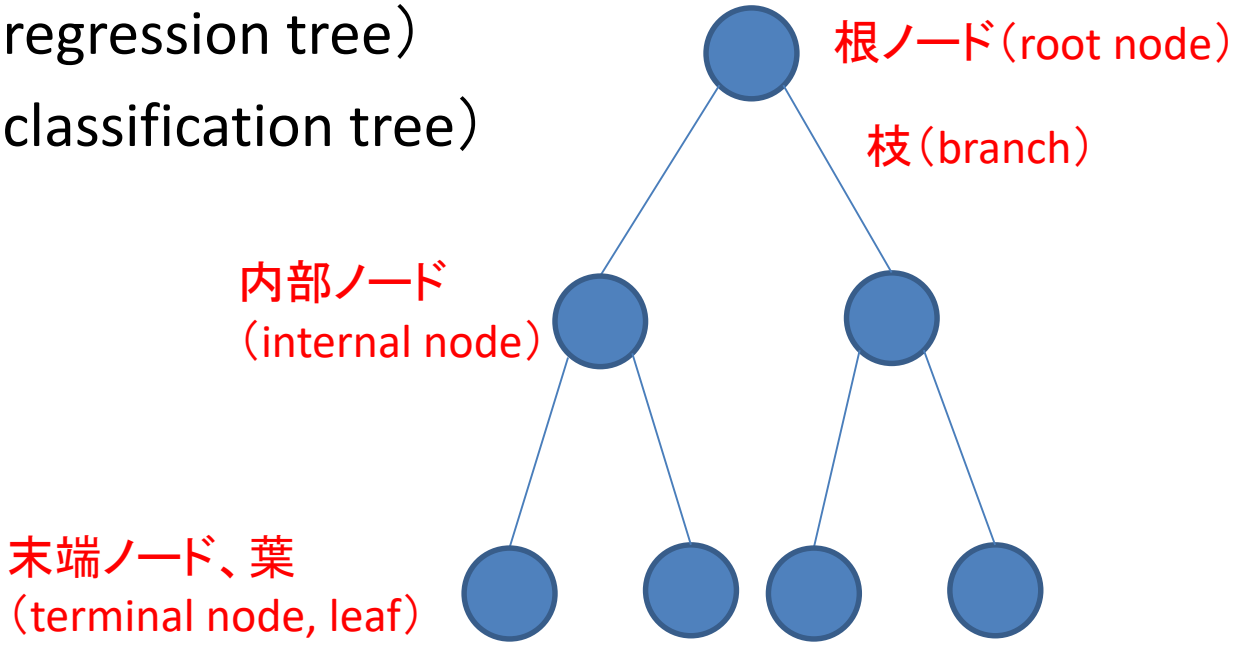
# 決定木

# 決定木

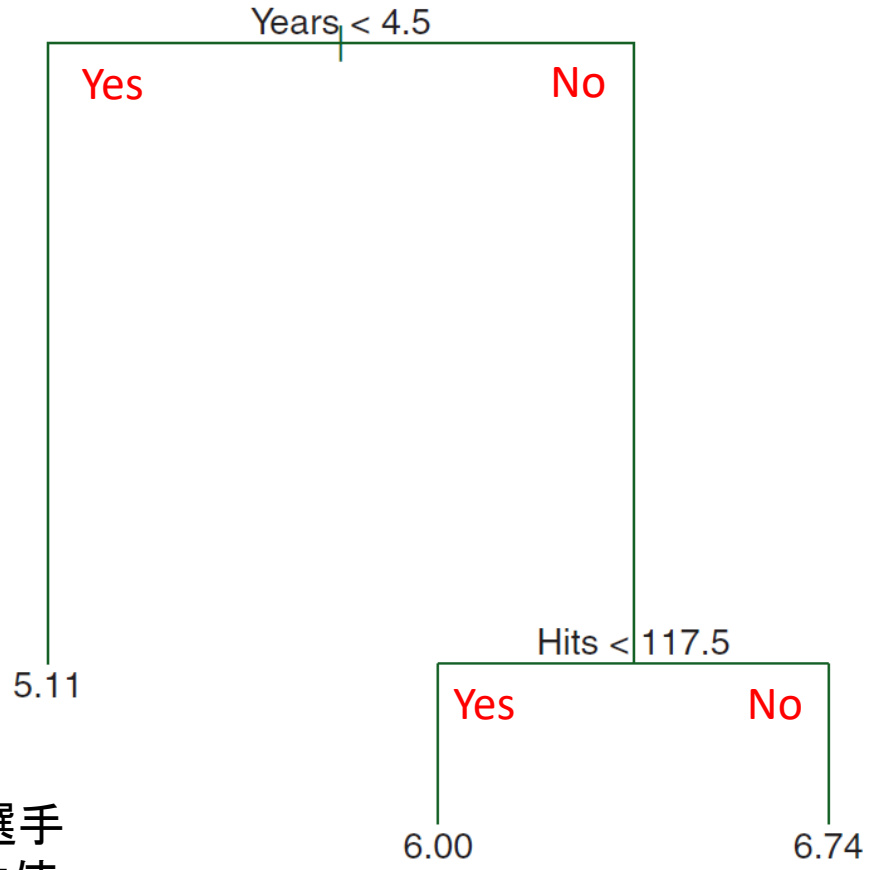
- 決定木（decision tree）
  - 木構造によって回帰や分類のための条件を表す
  - 種類
    - 回帰木（regression tree）
    - 分類木（classification tree）



# 回帰木

- 例
  - Hitters データセット
  - Years と Hits から Log(Salary)を予測

Years: 大リーグの経験年数
Hits: 昨年のヒットの数




Years が 4.5 未満の選手の Log(Salary) の平均値

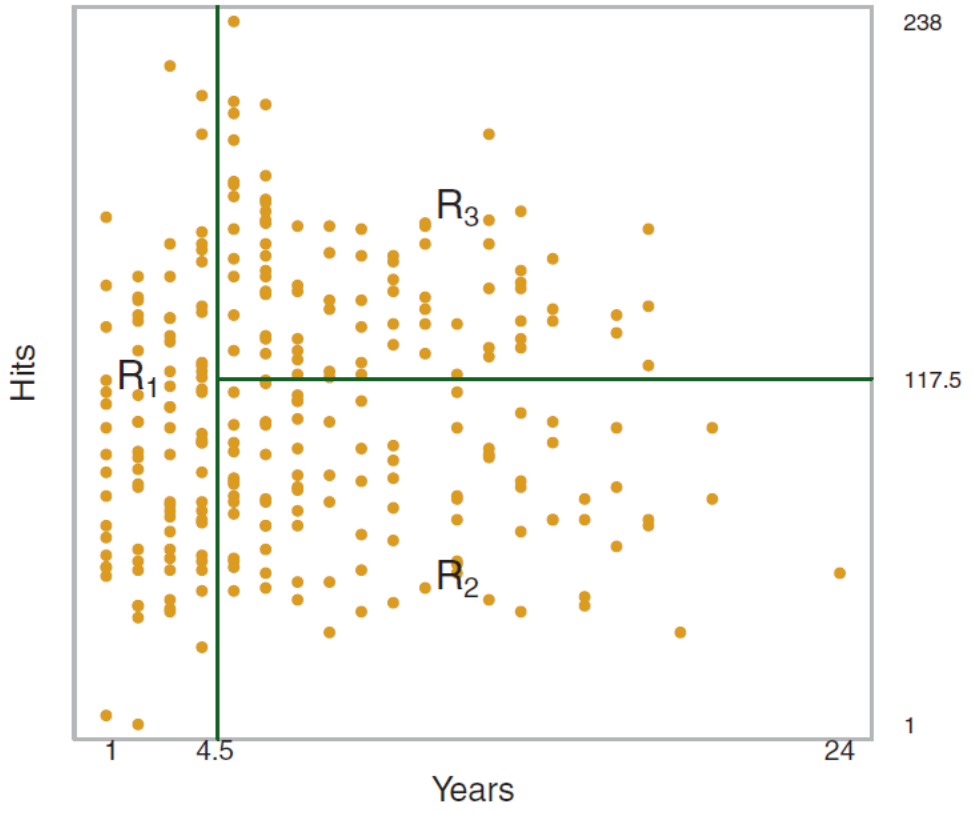
# 回帰木

- 予測変数の空間の区分け
  - $R_1$, $R_2$, $R_3$: 終端ノード

$$R_1 = \{\mathrm{X} \mid \texttt{Years<4.5}\}$$
$$\$1{,}000 \times e^{5.107} = \$165{,}174$$

$$R_2 = \{\mathrm{X} \mid \texttt{Years>=4.5, Hits<117.5}\}$$
$$\$1{,}000 \times e^{5.999} = \$402{,}834$$

$$R_3 = \{\mathrm{X} \mid \texttt{Years>=4.5, Hits>=117.5}\}$$
$$\$1{,}000 \times e^{6.740} = \$845{,}346$$



# 回帰木

- 回帰木の作り方と使い方
  1. 予測変数の空間を $J$ 個の領域 $R_1, R_2, \ldots, R_J$ に区分け
  2. 予測変数の値が領域 $R_j$ に含まれる入力に対し、$R_j$ に含まれる学習データの応答変数の平均値を出力

- 区分けの方法
  - 学習データへのフィット

$$\mathrm{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} \left( y_i - \hat{y}_{R_j} \right)^2$$

$\hat{y}_{R_j}$ : 領域 $R_j$ に含まれるデータの応答変数の平均値

を最小化する区分けを求めればよさそうだが計算量的に無理

# 回帰木

- 貪欲法で再帰的に区分け
  - 分割
    - 全ての予測変数 $X_1, X_2, \ldots, X_p$ と分割の基準値 $s$ の組み合わせを考え

$$\sum_{i:x_i \in R_1(j,s)} \left(y_i - \hat{y}_{R_1}\right)^2 + \sum_{i:x_i \in R_2(j,s)} \left(y_i - \hat{y}_{R_2}\right)^2$$

$$R_1(j,s) = \left\{X \middle| X_j < s\right\}$$

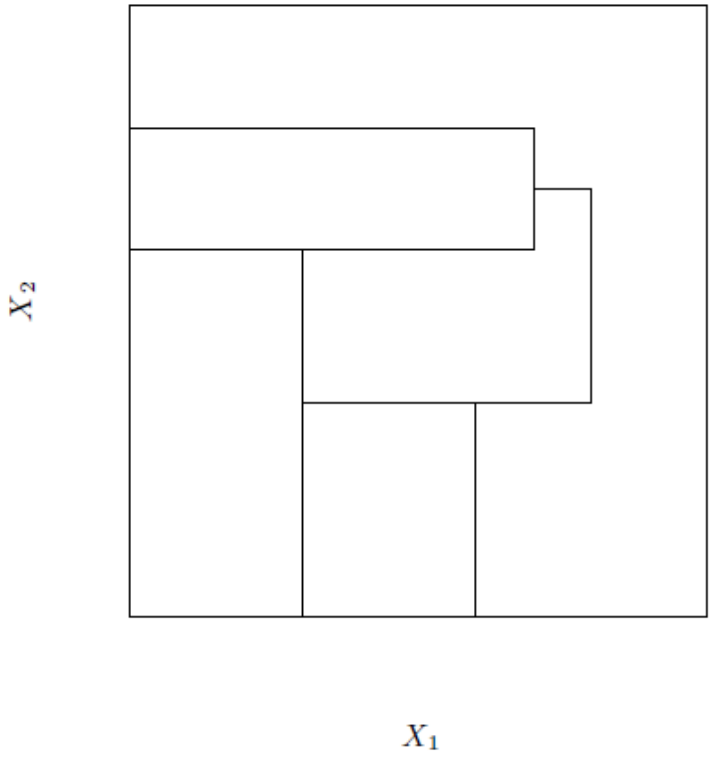
$$R_2(j,s) = \left\{X \middle| X_j \geq s\right\}$$

      が最小になる分割方法を選択

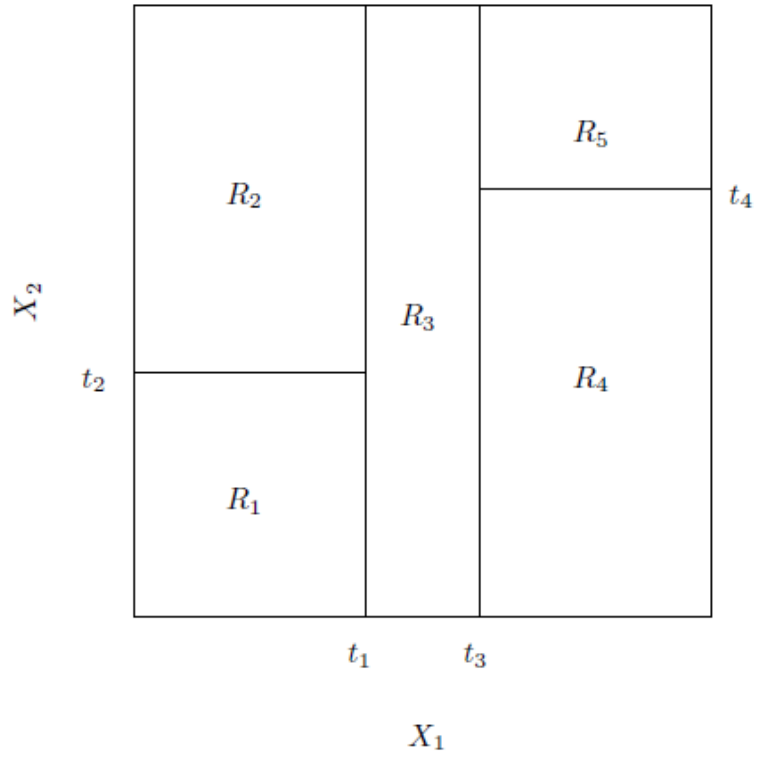
  - 再帰
    - 分割によって作られた2つの領域をさらに同様に分割
    - 領域に含まれる事例の数が閾値以下になるまで繰り返す

# 回帰木

再帰的な分割では不可能な区分け

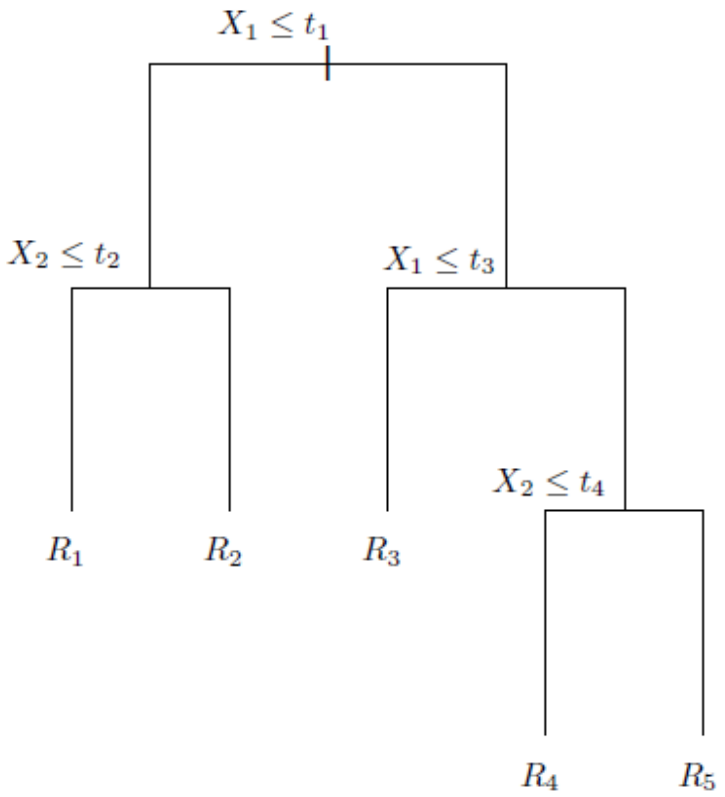



再帰的な分割で可能な区分け

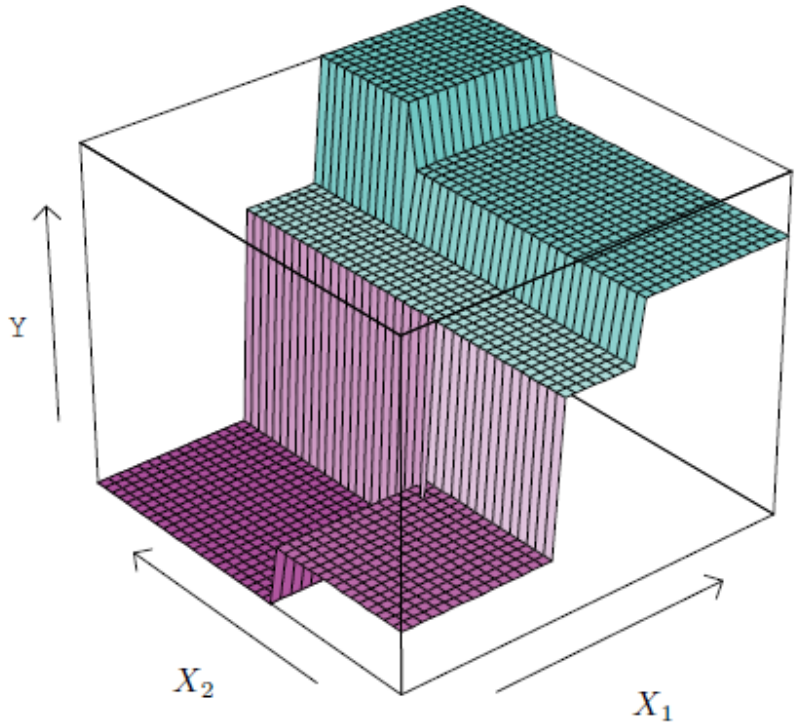


# 回帰木

前ページ右図に対応する回帰木

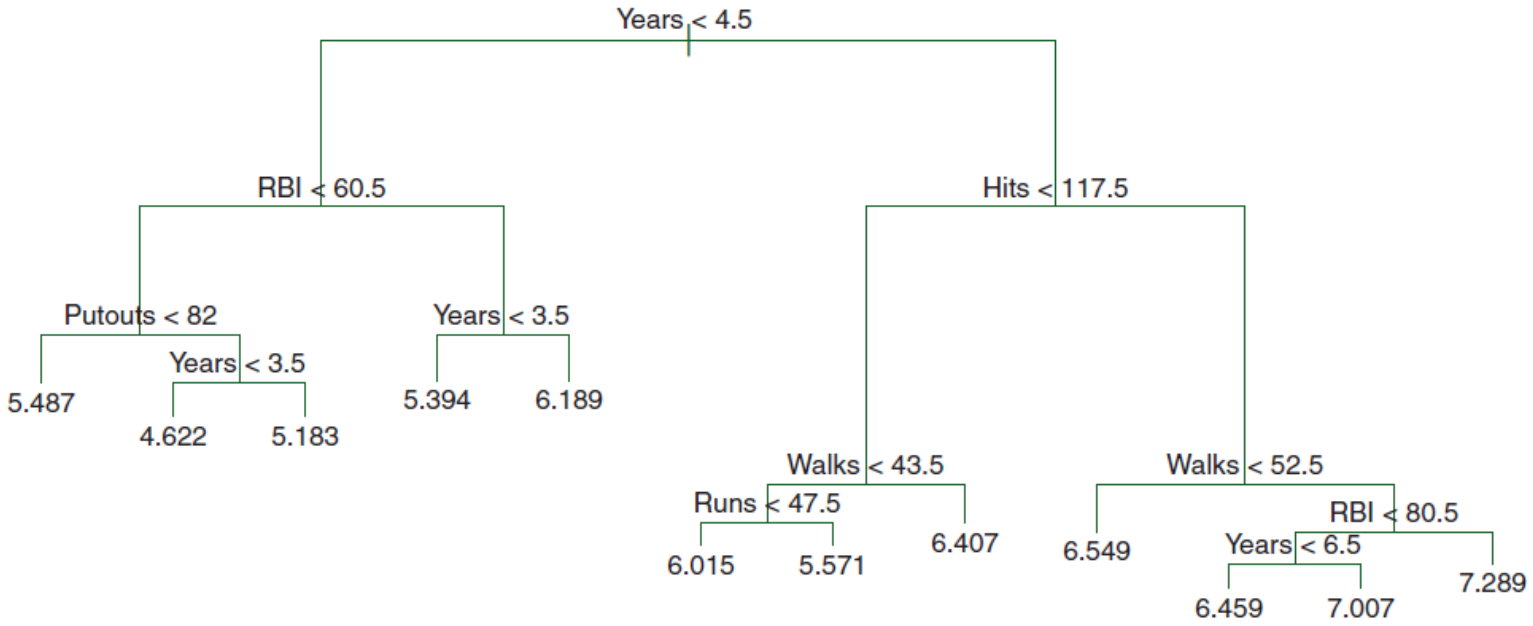



予測変数の空間と出力値



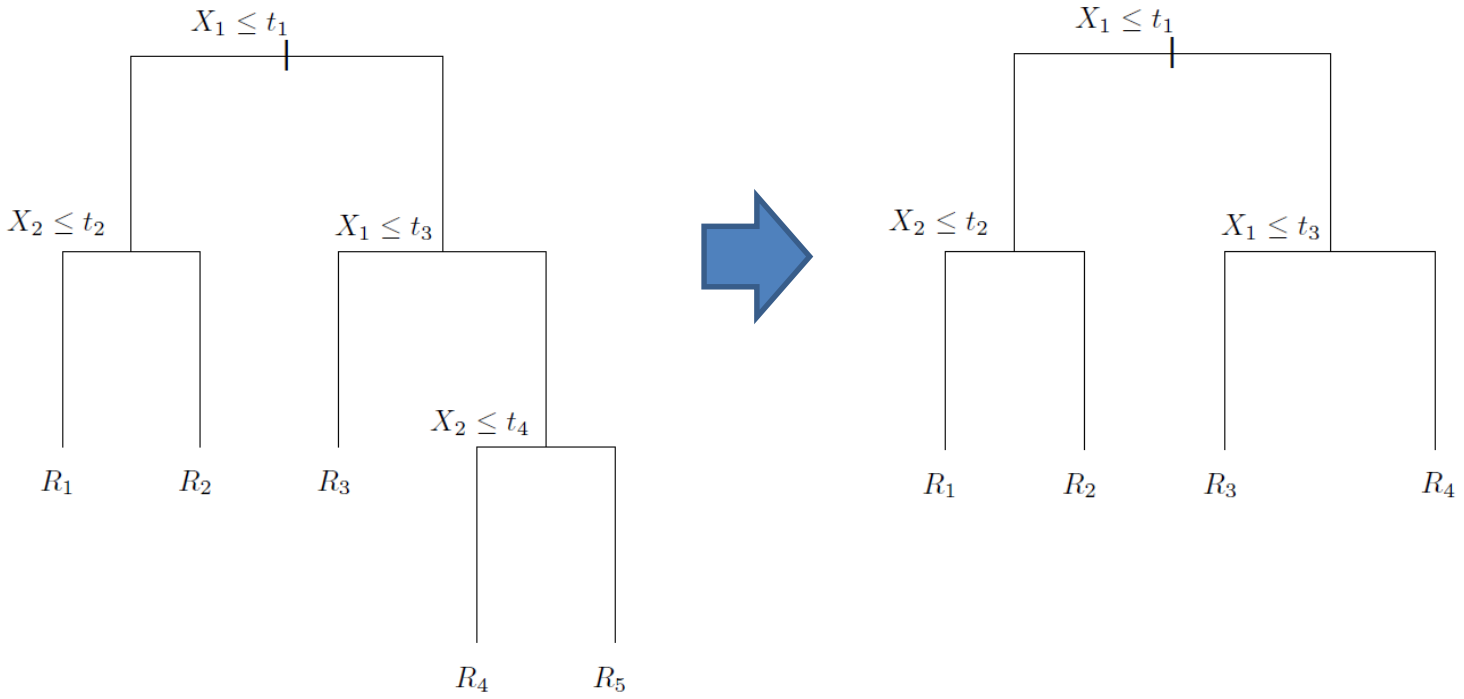
# 回帰木

- Hitters データに対する回帰木



# 回帰木

- 枝刈り（pruning）
  - 木のサイズを小さくして過学習を抑制



# 回帰木

- コスト複雑度枝刈り（cost complexity pruning）
  - $\alpha$: 木の複雑さに対するペナルティの重み
  - 与えられた $\alpha$ に対して

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} \left(y_i - \hat{y}_{R_m}\right)^2 + \alpha |T|$$

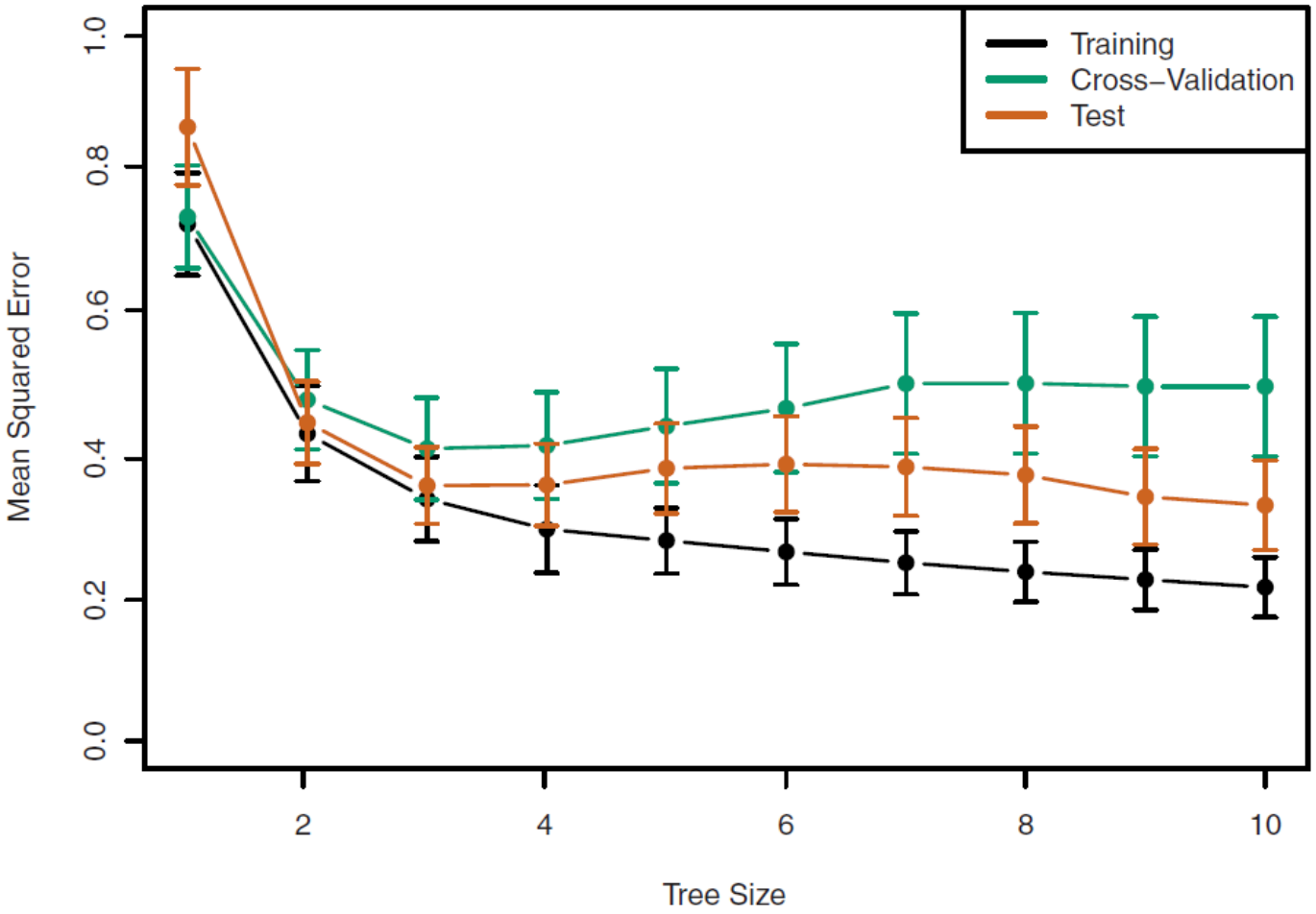
末端ノードの数

が最小になるように枝刈り

  - 交差検証で $\alpha$ をチューニング

# 回帰木

- 木のサイズとMSE（Hitters データ）



# 分類木

- 分類木
  - 応答変数が質的変数
  - 末端ノードでの出力は平均の代わりに多数決

- 構築法
  - 回帰木と同様、再帰的な分割を繰り返す
  - 分割に用いる変数と値を決める指標
    - 分類誤り率ではなく Gini index や entropy などがよく用いられる

$$G = \sum_{k=1}^{K} \hat{p}_{mk}\left(1 - \hat{p}_{mk}\right) \qquad D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

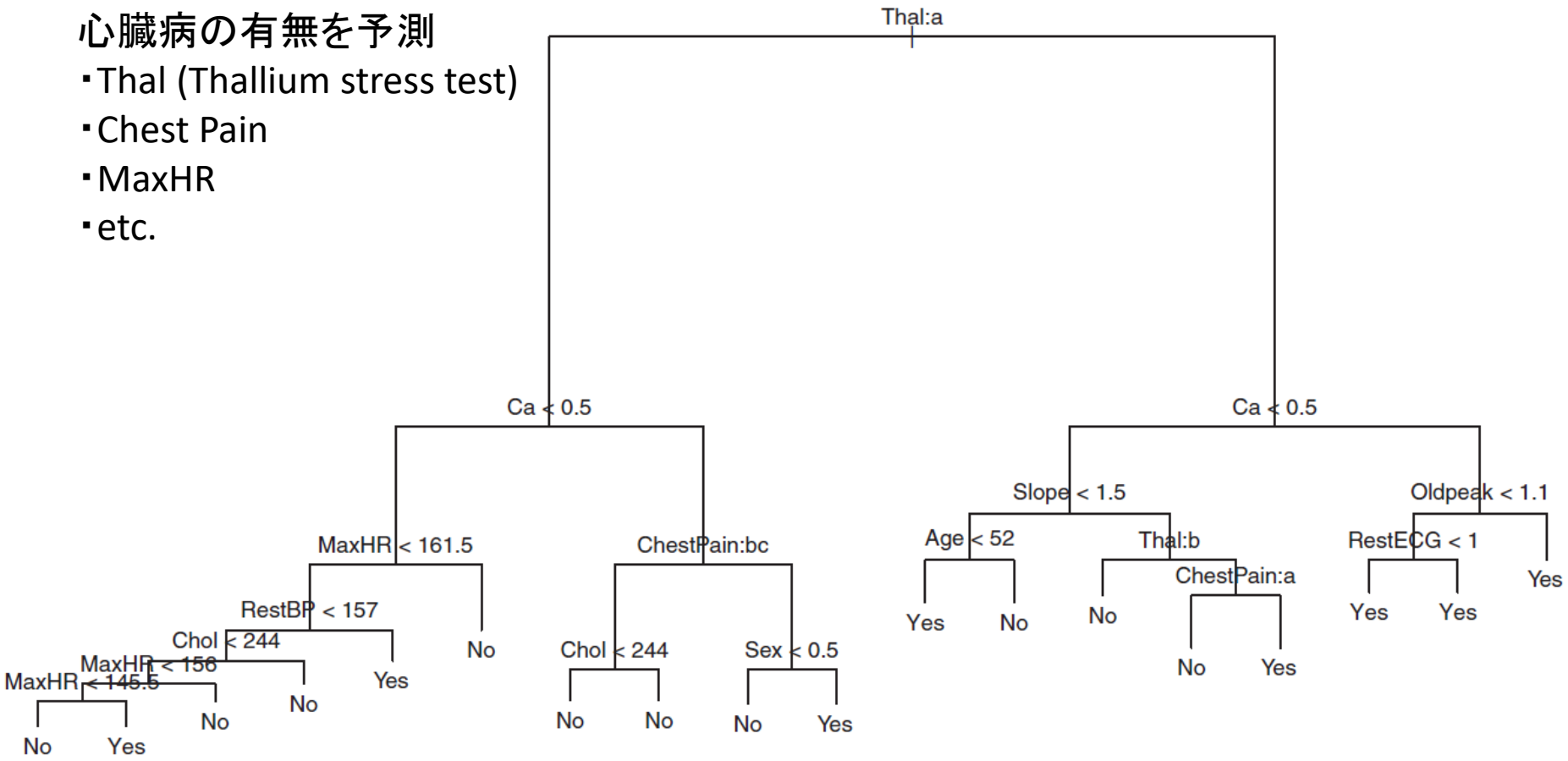
$\hat{p}_{mk}$：領域 $m$ に含まれるクラス $k$ の事例の割合

# 分類木

- Heart データセット

心臓病の有無を予測
- Thal (Thallium stress test)
- Chest Pain
- MaxHR
- etc.

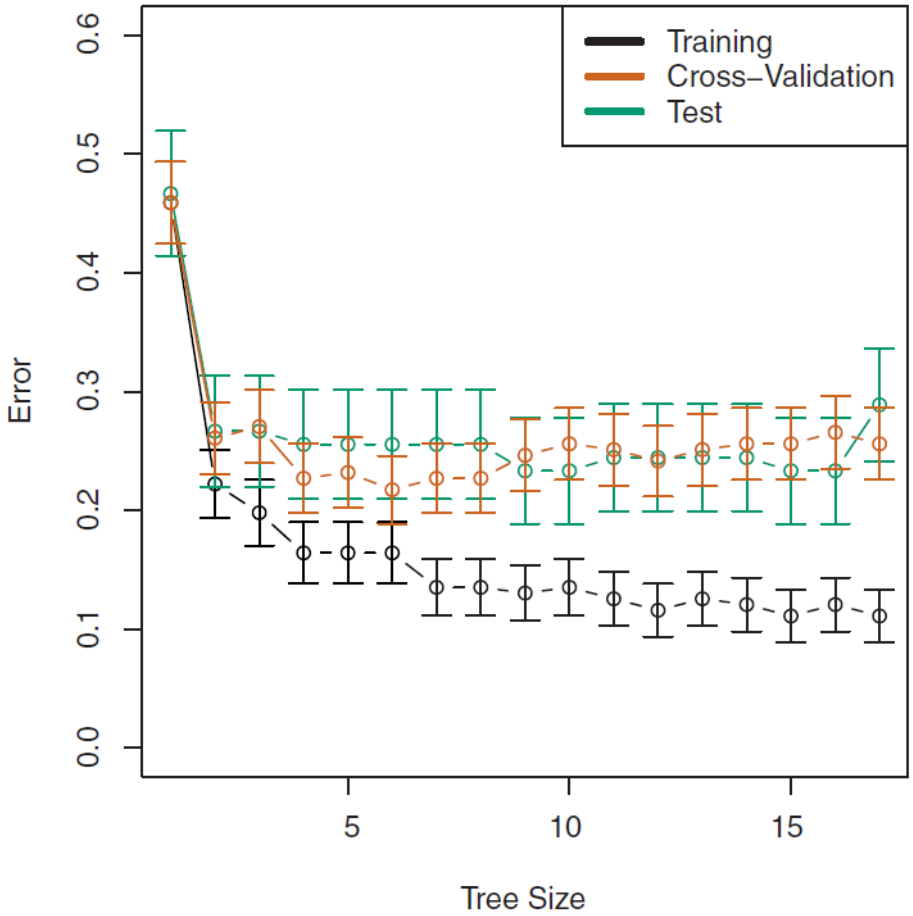
Thal（質的変数）
a: normal
b: fixed
c: reversible defects

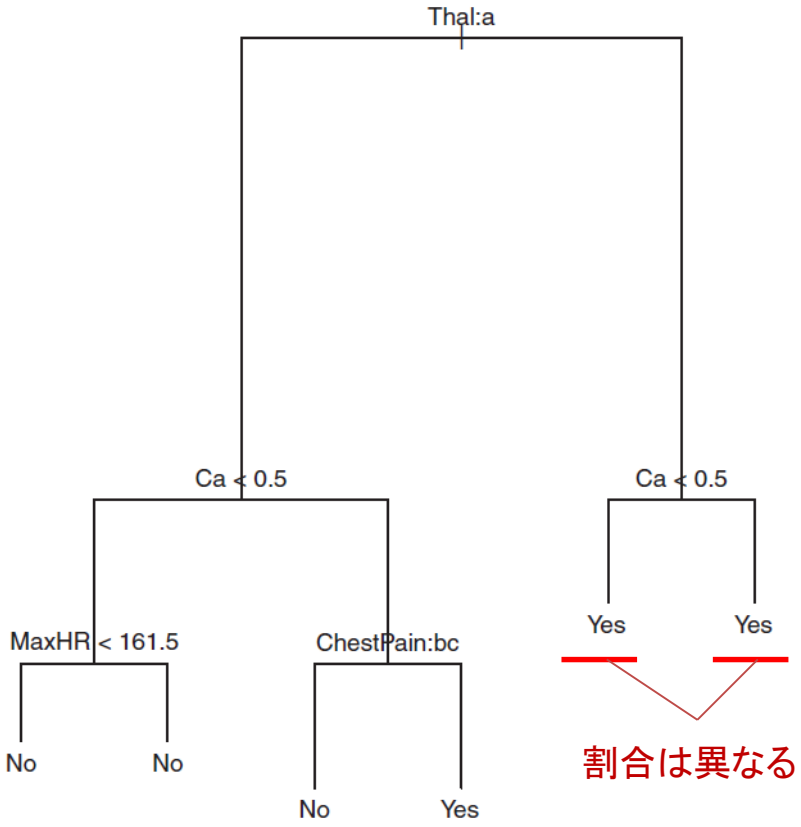


# 分類木

- 枝刈り

交差検証エラーが最小の木



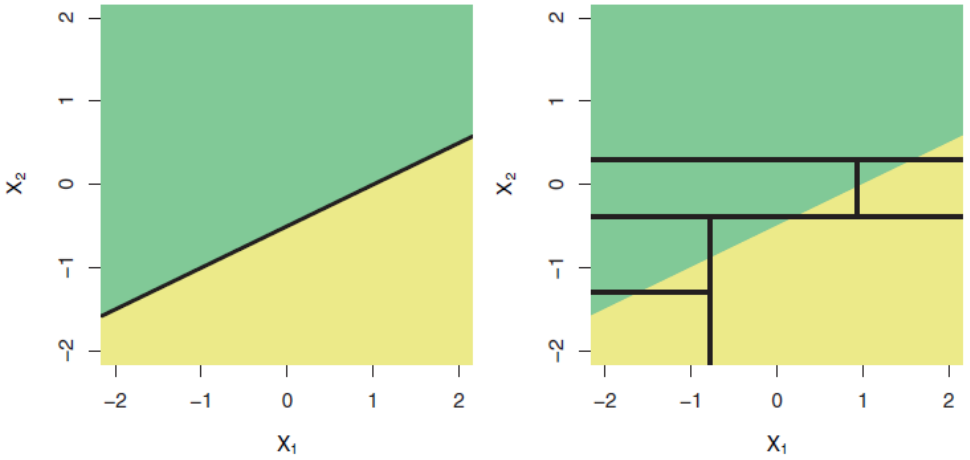



# 決定木 vs 線形モデル

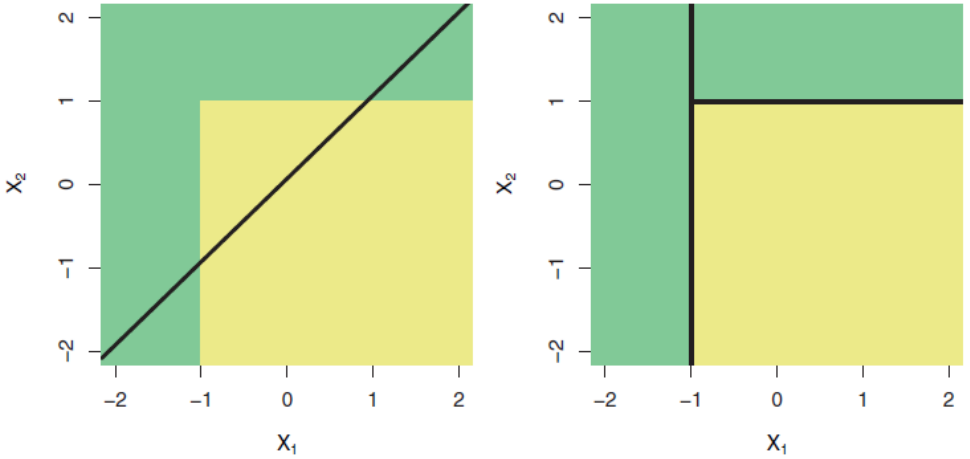
- 例

線形モデル　　決定木

真のデータ分布が
線形モデルで表現
可能な場合




真のデータ分布が
線形モデルで表現
できない場合



# 決定木

- 決定木の長所と短所
  - 長所
    - 学習結果の解釈が容易
    - 人間の意思決定の方法と近い（と考える人もいる）
    - 質的変数を扱う際にダミー変数を作らなくてよい
  - 短所
    - 回帰や分類の精度があまり高くない
    - 学習データの小さな違いで構築される木が大きく変わることがある

# 準備

- Windows の場合
  - コマンドプロンプトからモジュールのインストール

```
> py -m pip install graphviz
> py -m pip install pydotplus
> py -m pip install jupyter
```

# Python実習

- 準備
  - Carseats データ
    - チャイルドシートの売り上げデータ

```
>>> import pandas as pd
>>> import os
>>> os.chdir(os.path.expanduser("~"))
>>> carseats = pd.read_csv('./Carseats.csv')
```

  - 売上(Sales)が8より大きいかどうかを示す変数を追加
  - カテゴリ変数を数値に整数に変換

```
>>> carseats.head()
>>> carseats['High'] = carseats.Sales.map(lambda x: 'Y' if x>8 else 'N')
>>> carseats.ShelveLoc = pd.factorize(carseats.ShelveLoc)[0]
>>> carseats.Urban = carseats.Urban.map({'No':0, 'Yes':1})
>>> carseats.US = carseats.US.map({'No':0, 'Yes':1})
>>> carseats.head()
>>> carseats.info()
```

# Python実習

- 分類木の構築
  - DecisionTreeClassifier()
  - Sales 以外の情報を使って High を予測

```
>>> X = carseats.drop(['Sales', 'High'], axis=1)
>>> y = carseats.High
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=200)
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(criterion = "gini", max_depth=6,
min_samples_leaf=4)
>>> clf.fit(X_train, y_train)
>>> clf.score(X_train, y_train)
```

# Python実習

- 木構造の表示
  - export_graphviz()

```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(clf, out_file ="mytree.dot", feature_names
= X_train.columns)
>>> with open("mytree.dot") as f:
...     dot_graph = f.read()
>>> import graphviz
>>> graphviz.Source(dot_graph)
```

# Python実習

- 精度評価

```
>>> y_pred = clf.predict(X_test)
>>> from sklearn.metrics import confusion_matrix, accuracy_score
>>> cm = pd.DataFrame(confusion_matrix(y_test, y_pred).T, index=['No', 'Yes'],
columns=['No', 'Yes'])
>>> cm
>>> accuracy_score(y_test,y_pred)*100
```

# Python実習

- Bostonデータセットの読み込み

```
>>> from sklearn.datasets import load_boston
>>> b = load_boston()
>>> boston = pd.DataFrame(b.data, columns=b.feature_names)
>>> boston.head()
...
>>> boston.shape
(506, 13)
```

# Python実習

- Boston データセット
  - ボストン郊外の住宅価格データ

| | crim | zn | indus | chas | nox | rm | age | dis | rad | tax |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.00632 | 18 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.09 | 1 | 296 |
| 2 | 0.02731 | 0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242 |
| 3 | 0.02729 | 0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242 |
| 4 | 0.03237 | 0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222 |
| 5 | 0.06905 | 0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222 |
| 6 | 0.02985 | 0 | 2.18 | 0 | 0.458 | 6.43 | 58.7 | 6.0622 | 3 | 222 |
| 7 | 0.08829 | 12.5 | 7.87 | 0 | 0.524 | 6.012 | 66.6 | 5.5605 | 5 | 311 |
| 8 | 0.14455 | 12.5 | 7.87 | 0 | 0.524 | 6.172 | 96.1 | 5.9505 | 5 | 311 |
| 9 | 0.21124 | 12.5 | 7.87 | 0 | 0.524 | 5.631 | 100 | 6.0821 | 5 | 311 |

  - CRIM: per capita crime rate by town
  - :
  - LSTAT: lower status of population (percent)
  - MEDV: median value of owner-occupied homes in $1000s

# Python実習

- 回帰木の構築
  - Boston データセット
    - medv: Median value of owner-occupied homes in $1000's
    - lstat: % lower status of the population
    - rm: average number of rooms per dwelling
    - :

```
>>> X = boston
>>> y = b.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.5, random_state=0)
>>> from sklearn.tree import DecisionTreeRegressor
>>> regr_tree = DecisionTreeRegressor(max_depth=2)
>>> regr_tree.fit(X_train, y_train)
```

# Python実習

- 回帰木のプロット

```
>>> export_graphviz(regr_tree, out_file = "mytree.dot",
feature_names = X_train.columns)
>>> with open("mytree.dot") as f:
...     dot_graph = f.read()
>>> graphviz.Source(dot_graph)
```

- 精度評価

```
>>> y_pred = regr_tree.predict(X_test)
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

# バギング

- バギング（bagging, bootstrap aggregating）
  - ブートストラップによってバリアンスを減らす
  - 手順
    - For $b = 1$ to $B$
      - 学習データから $n$ 個の事例をランダムに復元抽出
      - 抽出された $n$ 個の事例を学習データとして（枝刈り無しの）回帰木を構築
    - 出力は $B$ 個の回帰木の出力の平均

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

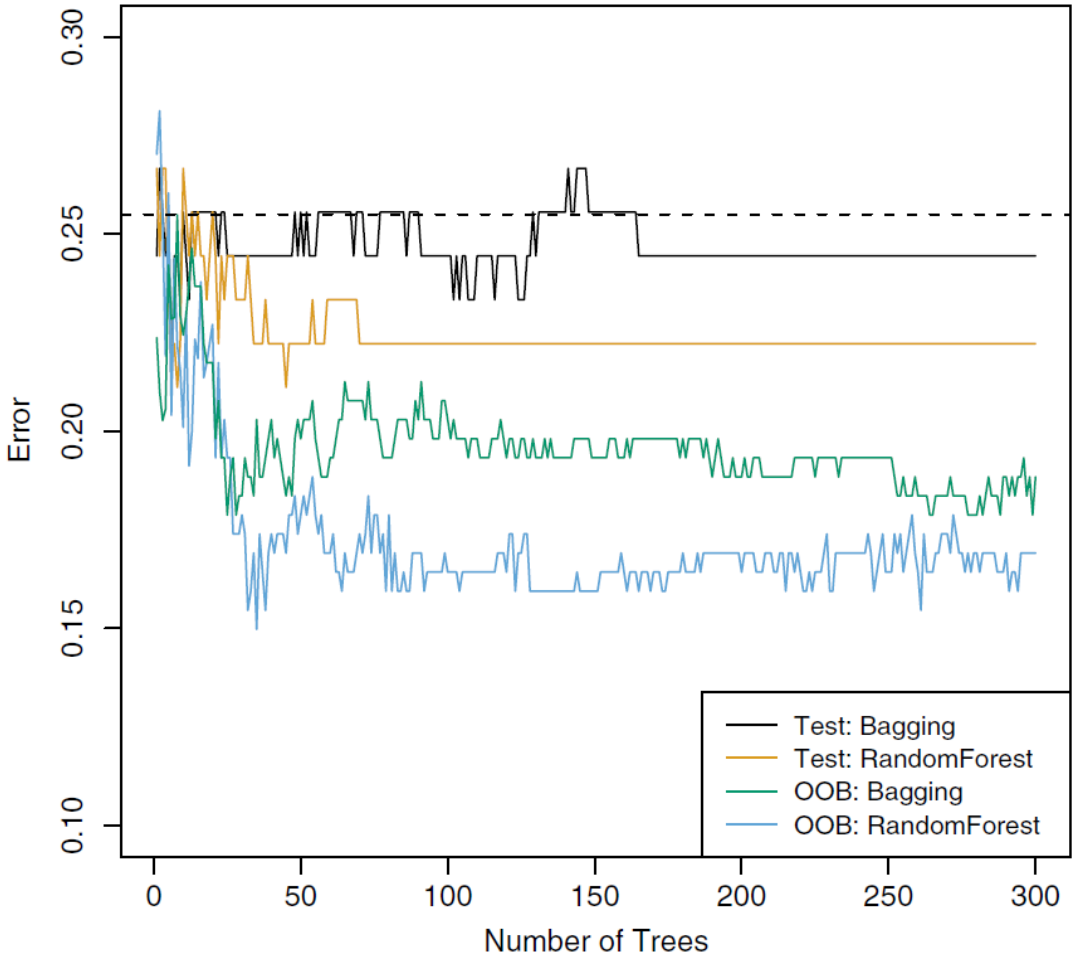
  - 分類木の場合は平均の代わりに多数決

# バギング

- Out-of-bag (OOB) エラー推定
  - テストエラーの効率的な推定法
  - 手順（回帰の場合）
    - For $i = 1$ to $n$
      - $B$ 個の回帰木のうち $i$ 番目の事例を学習に使わなかった回帰木の出力の平均を予測値とし、$y_i$ と比較
  - LOO交差検証とほぼ同等の推定結果が交差検証を行わずに得られる

# バギング

- Heart データでのテストエラー



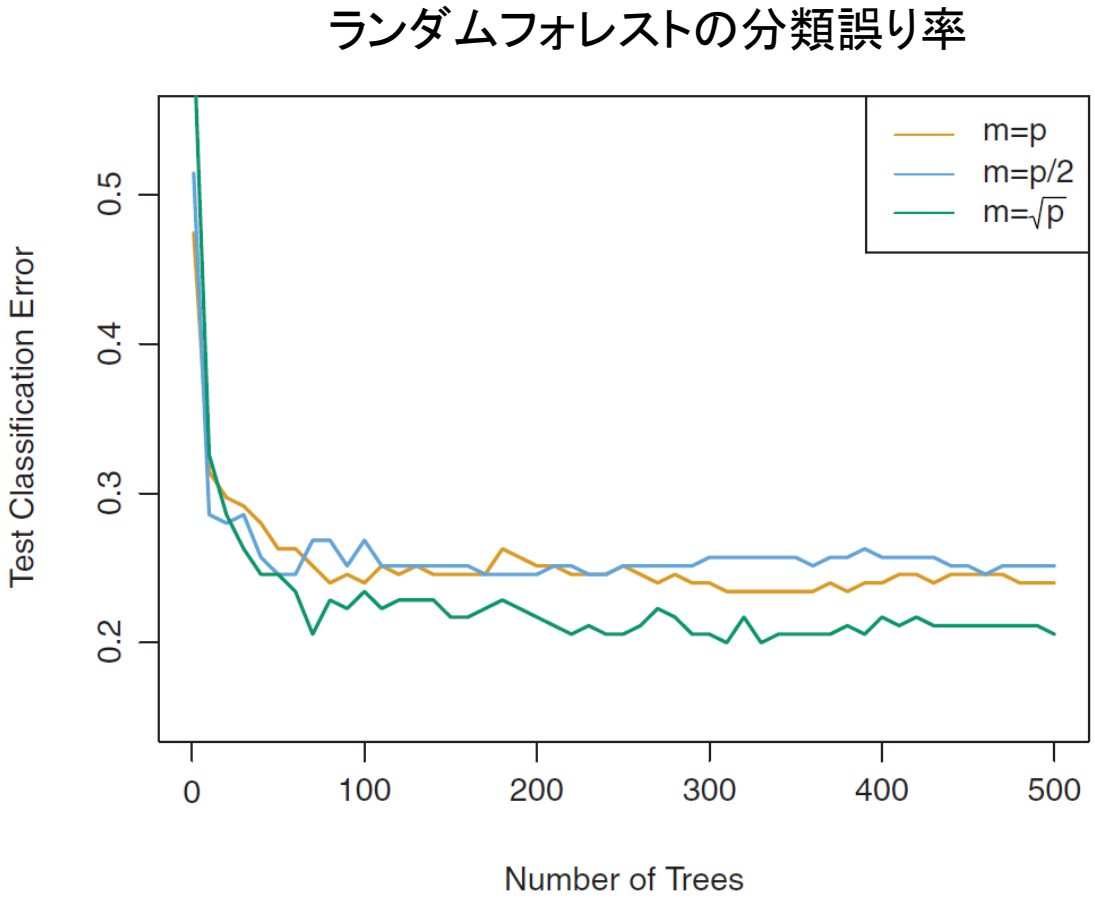

単一の分類木
のテストエラー

# ランダムフォレスト

- ランダムフォレスト（random forest）
  - 決定木を構築するための再帰的な分割の際に考慮する予測変数を制限すること以外はバギングと同じ
  - 各分割で考慮する予測変数を（分割ごとに）ランダムに選択された $m$ 個の予測変数に限定
    - $m = p$ であればバギングと同じ
    - $m = \sqrt{p}$ がよく用いられる
  - 単純なバギングよりも構築される決定木のバリエーションが増える
    - 構築された決定木の間の相関が減る

# ランダムフォレスト

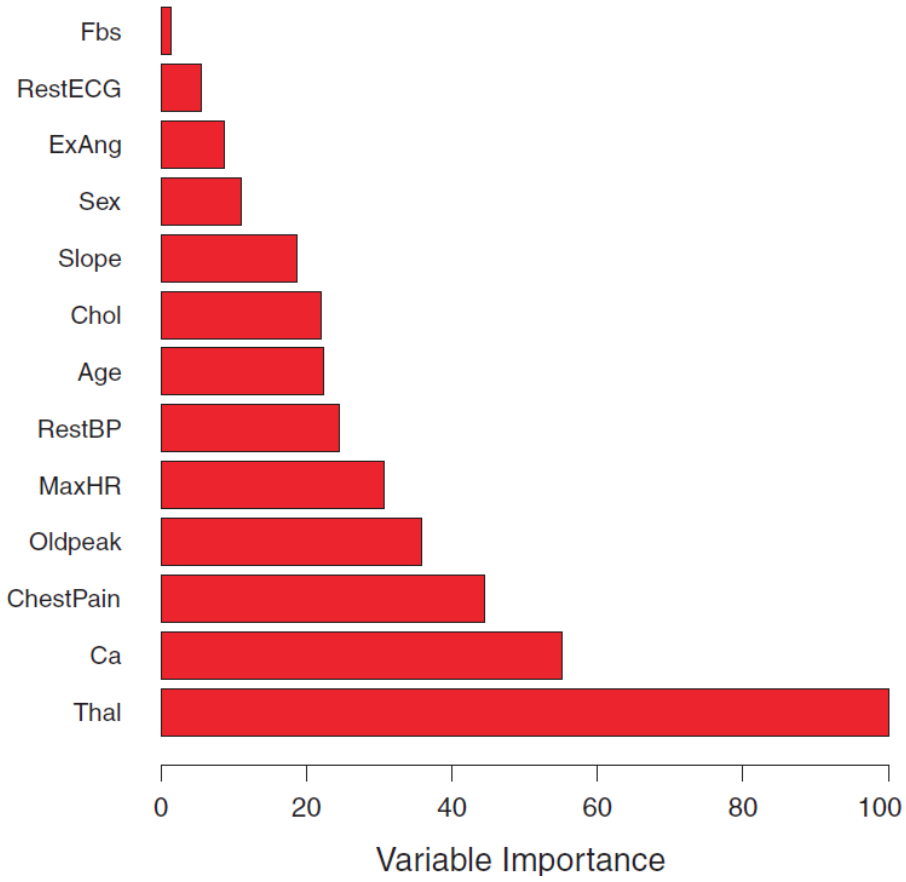
- 遺伝子の発現からがんの種類を予測
  - 349人の患者
  - 応答変数：15クラス
    - Normal + 14種類のがん
  - 予測変数
    - 500遺伝子の発現量
- 単一の分類木の分類誤り率は45.7%



# バギング

- バギングやランダムフォレストは精度を大きく向上させることができる反面、モデルの解釈が難しくなる

- 変数の重要度
  - 各変数による分割によって、どれだけ RSS（回帰の場合）や Gini index（分類の場合）が減少したか



# ブースティング

- ブースティング（boosting）
  - 学習器を逐次的に多数構築して学習データにフィッティング
  - バギング同様、決定木に限らず様々なモデルに適用可能

- 回帰木のブースティング
  - （小さな）回帰木を逐次的に多数構築
  - 残差を応答変数として回帰木を作る
  - 最終的なモデルの出力
    - 構築された多数の回帰木の出力の和

# ブースティング

手順

- 初期化

$$\hat{f}(x) = 0, \quad r_i = y_i$$

- For $b = 1$ to $B$
  - 分割数が $d$ の決定木 $\hat{f}^b$ を学習データ $(X, r)$ から構築
  - 構築された木によって $\hat{f}$ を更新

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

  - 残差を更新

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x)$$

- モデルを出力

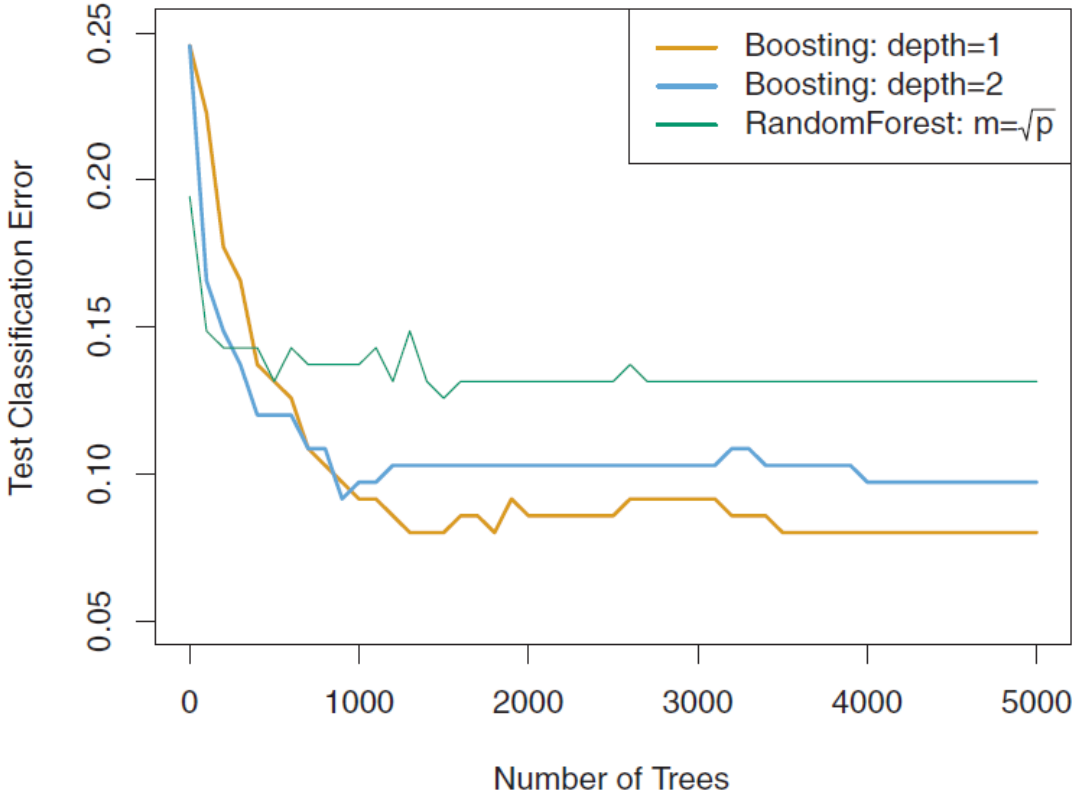
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

# ブースティング

- 遺伝子の発現からがんの種類を予測
  - 2値分類
    - Normal vs がん
- 単一の分類木の分類誤り率は24%

$\lambda = 0.01$

$d = 1 \text{ or } 2$



# Python実習

- バギング
  - RandomForestRegressor()
    - max_features 引数で分割の際に考慮する変数の数を指定
      - max_features = $p$ とすれば全ての変数を使用するので単純なバギング

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> all_features = X_train.shape[1]
>>> regr_bagging = RandomForestRegressor(max_features=all_features)
>>> regr_bagging.fit(X_train, y_train)
```

- 精度評価

```
>>> y_pred = regr_bagging.predict(X_test)
>>> mean_squared_error(y_test, y_pred)
```

→ 単一の回帰木よりもかなり良い

# Python実習

- ランダムフォレスト
  - `RandomForestRegressor()`
    - max_features の値を *p* よりも小さな数にする

```
>>> regr_rf = RandomForestRegressor(max_features=4)
>>> regr_rf.fit(X_train, y_train)
>>> y_pred = regr_rf.predict(X_test)
>>> mean_squared_error(y_test, y_pred)
```

→ 単純なバギングからさらに向上

# Python実習

- ブースティング
  - GradientBoostingRegressor()

```
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> regr_boost = GradientBoostingRegressor(n_estimators=500,
learning_rate=0.02, max_depth=4)
>>> regr_boost.fit(X_train, y_train)
```

# Python実習

- 変数の重要度

```
>>> feature_importance = regr_boost.feature_importances_*100
>>> rel_imp = pd.Series(feature_importance,
index=X_train.columns).sort_values(inplace=False)
>>> rel_imp.T.plot(kind='barh', color='r', )
>>> plt.xlabel('Variable Importance')
>>> plt.gca().legend_ = None
>>> plt.show()
```

- 精度評価

```
>>> y_pred = regr_boost.predict(X_test)
>>> mean_squared_error(y_test,y_pred)
```