The Elements of Statistical Learning
Chap.10: Boosting and Additive Tree

Kosuke Kito

August 30, 2020

流れ

- “AdaBoost.M1” の話
- Boosting と Additive Model の相性がいい話
- Forward Stagewise(Additive Model で使える計算方法) の話
- 損失関数の話
- Data Mining における手法の “実用性” の話
- 木を使った Boosting の話
- 勾配 Boosting の話
- 正則化の話
- 計算結果の解釈の話

Boosting とは？ざっくりと.

- “弱い” 分類器を組み合わせて使うことで, 効果的な分類を行う手法.
- この元になる　分類器たちのことを “base learner” と呼ぶ.
- 元々は分類用に考えられた. 今は回帰にも応用されている.
- bagging との違いは, 前の手順の推定結果が次の手順に影響するかしないか. (袋から球を取り出した後, 戻すか戻さないかのイメージ?)
- 例として, Boosting で一番有名な “AdaBoost.M1” というアルゴリズムを紹介するよ.

AdaBoost.M1

- 2 クラスへの分類問題用のアルゴリズム. (当面は 2 クラスの分類問題しか考えません. が, 多値分類や回帰にも自然に拡張できる感じです.)
- 各 G_m $(m = 1, \ldots, M)$ は, 弱い分類器. つまり, $\{-1, 1\}$ への関数で, 正解率がランダムよりは多少良いもの.
- それまでの手順で正しく分類できなかったデータ程, 重視される. (画像の 2-(d))
- 最後に合算するときは, 正答率の高かった時の推定が重視される (画像の 3)
- 各分類器の値域が離散値なので, "Discrete AdaBoost" とも.
- 値域を連続値 $[-1, 1]$ に変更して, 確率の推定に使うこともできる.

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

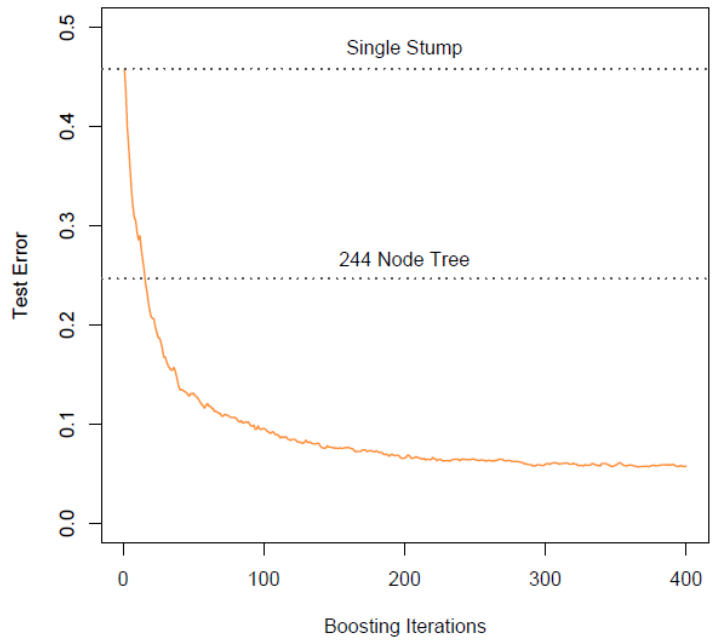
AdaBoost.M1

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Boosting すごいぞ

- 用語. “Stump” とは, 単純な 2 分類の分類木.
- もちろん, とても弱い分類器.
- でも, これで Boosting すると, 244 分類の分類木よりも良い性能を出せる.



Boosting と Additive Model

AdaBoost の最後の式と Additive Model の式は似ている.

- AdaBoost の最後の式

$$G(x) = \mathrm{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

- Additive Model の式

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

→ Boosting は, base learner を使った加法的な基底関数展開と思える.
こんな感じで, Additive Model の特殊な例と思えるものが, この本の中には多い.

Additive Model の最適化

Additive Model の最適化は, 損失関数を適当にとって, 損失の合計を最小化しようとすることが多い.

$$\min_{\{\beta_m, \gamma_m\}_{m=1}^{M}} \sum_{i=1}^{N} L\left(\sum_{m=1}^{M} \beta_m b(x; \gamma_m)\right)$$

ただし, 計算量が凄いことになるので, 困る. そこで出てくるのが, "Forward Stagewise Additive Modeling".

Forward Stagewise Additive Modeling

- いわゆる, 貪欲法 (Greedy algorithm).
- 各基底関数ごとの最適化は簡単なときに有効.
- 前から一個ずつそこまでの結果との和を使って最適化していく.
- Squared-error loss

$$L(y, f(x)) = (y - f(x))^2$$

に代表される, 残差にのみ依存する損失関数の場合, 残差に向かう最適化と思っても同義.

(ちなみに, Squared-error は, 分類のための損失関数として良い選択肢じゃないよ.
ちょっと後で考える.)

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute
   $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$
   (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.

Forward Stagewise と AdaBoost

- AdaBoost.M1 は, 損失関数として以下を使った Forward Stagewise Additive Modeling と思える.

$$L(y, f(x)) = e^{-yf(x)}$$

- 証明は割愛. 結構単純な計算なので, やってみるとよいかも？
- 損失関数として, これを採用する理由については後述.
- ちなみに, AdaBoost は, Forward Stagewise とは全然別に考えられて, 後から一緒やん, ってなったらしいです.

指数損失関数を使うことの正当化

- 指数損失関数の強みの一つは, 計算が簡単なこと. AdaBoost の反復計算の重みの更新はとても単純.
- 一方, 統計的にも良い. 指数損失関数の"Population Minimizer" は, "Deviance" のそれと一致し, 母集団における事後確率になる.
- 用語. "Population Minimizer" とは, パラメータ付きの確率分布から実数への関数を最小化するパラメータの値. 前に送った, 平均値・中央値・最頻値の話が非常に良い例になっている.
  例えば, 平均値は, 期待二乗誤差の "Population Minimizer" である, と言える.
- 用語. "Deviance" とは, $-\log$(尤度) で定まる値であり, 最尤法の損失関数と思える関数である. (たぶん, 結構バックグラウンドの分厚い概念だと思うのですが, あまり把握しきれず.)
  Binimial Deviance(2 値分類における Deviance) は, 以下の式で定まる.

$$\log\left(1 + e^{-2Yf(x)}\right)$$

指数損失関数を使うことの正当化

- 再び, 2 個目の視点に戻る. 指数損失関数の"Population Minimizer"は, 実際, 指数損失関数を使って最適化した式を $f^*(x)$ とすると,

$$f^*(x) = \arg\min_f \mathrm{E}_{Y|x}[e^{-Yf(x)}] = \frac{1}{2}\log\frac{\mathrm{P}[Y=1 \mid x]}{\mathrm{P}[Y=-1 \mid x]}$$

  となることが容易に分かる. もちろん, "Deviance" を最小化するものは, 尤度を最大化するものなので, これは事後確率になって, 一致すると分かる.
- ちなみに, 上の式から, AdaBoost の最後に sgn 関数を使うことも納得.
- まとめると, 計算が楽 (前節より) で, 最尤法と同じような答えが期待できるので, 指数損失関数はいいんじゃないか, という話.

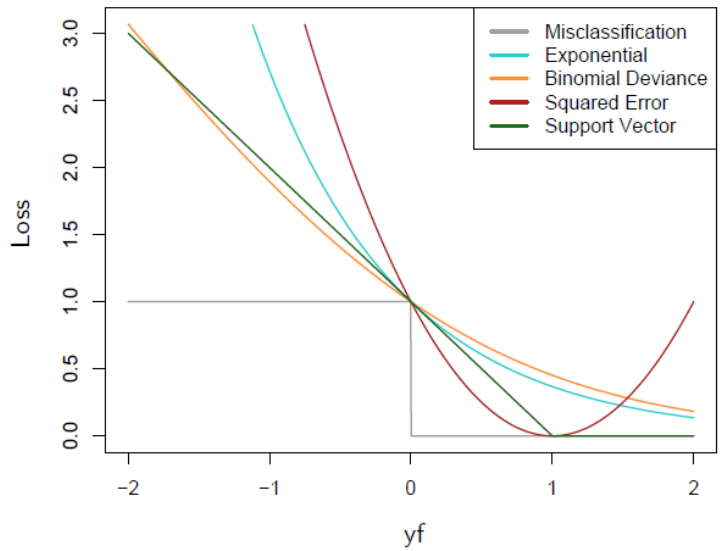
損失関数と頑健さ

様々な損失関数を, 分類と回帰のそれぞれの場合について, “頑健さ” (robustness) の視点から特徴づける.
頑健さ, とは, 訓練データに異常値が含まれているときに, それに影響を受けにくい度合いのこと. “noisy” な状況, つまり, 測定の質が低いときや, 母集団の分散が大きいときに, 頑健さが発揮される.
損失関数の頑健さは, y と $f(x)$ の値が離れるときの損失関数の値の大きくなる早さで評価できる.

分類における頑健さ

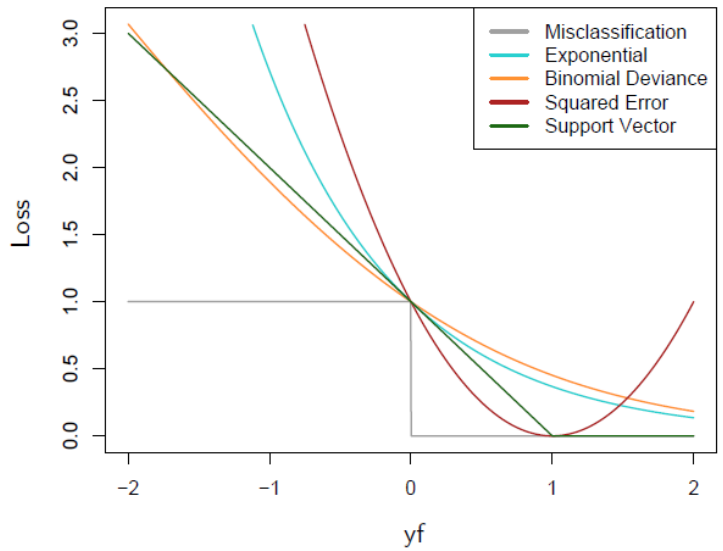
- 2 値 (-1/1) 分類では, 回帰における残差の代わりに積 $yf(x)$ を採用できる.
- 誤分類は不連続な 2 値関数. 他は誤分類を扱いやすく連続にしようとしているものと思える.
- 母集団全体について考えると, 指数損失関数と Deviance は同じ推定をするが, 有限の訓練データに対しては異なる.
  $yf << 0$ の時, 指数損失関数は指数関数的に増加するが, Deviance は線形であり, Deviance の方が頑健であるといえる.
  実際, AdaBoost は noisy な状況ではパフォーマンスが大幅に悪化するらしい.

ちなみに, ここら辺の損失関数たちとかは, 自然と多値分類問題に拡張できます.



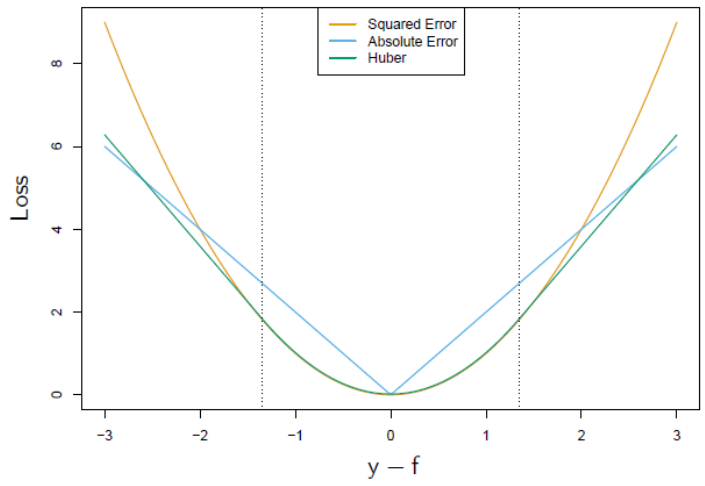
分類における頑健さ

- Squared-error は, 指数損失関数よりさらに発散が早いことが分かる. さらに, $yf(x) > 1$ のとき, 損失が増加していることもわかる. これらが, さっき「良い選択肢じゃない」といった理由.
- Squared-error の改善版として, ""Huberized" square hinge loss" というものがある. $|yf| > 1$ の部分を直線に変えたもの.
  計算も楽で, 頑健で, かなりいい感じらしい.



回帰における頑健さ

- 回帰では, 損失関数は残差 $y - f(x)$ の関数として定めればよい.
- Squared-error は平均, 残差の絶対値は中央値と対応することは前にまとめた通り.
  中央値の方が頑健であることもすぐに分かる.
- 中央値の頑健さと Squared-error の滑らかさを両立させる方法として, “Huber loss” というものがある.
  一定範囲外は直線, 範囲内は放物線.



頑健さと扱いやすさ

以上を踏まえると, 回帰において Squared-error を使ったり, 分類において指数損失を使うのは, 頑健さの観点から, 微妙なチョイス.
一方で, これらはとても計算しやすい. (双方, Forward Stagewise の 1 周期分の計算が非常に簡潔になる.)
→ 頑健さと扱いやすさを両立した方法を考えられないか？→ 単純に損失関数を頑健なものに変えるだけではできないので, 後の節で考えます.

実用性の観点と比較

- 入力値の形式が色々でも耐えるか. (numerical, binary, categorical)
- 欠損値
- 外れ値
- 入力値の単調変換
- 計算量
- 実は意味のなかった入力変数
- 入力変数の合体 (PCA のイメージ)
- 結果の解釈しやすさ
- 予測能力

TABLE 10.1. *Some characteristics of different learning methods. Key:* ▲*= good,* ◆*=fair, and* ▼*=poor.*

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of "mixed" type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

実用性の観点と比較

- 決定木は，予測能力以外とても良い方法．
- 決定木を使って Boosting すれば，予測能力を向上出来ていい感じ．
- 計算量，解釈しやすさが多少犠牲になる．
- これらの犠牲をさらに減らす方法として，勾配 Boosting が考えられている．

TABLE 10.1. *Some characteristics of different learning methods. Key:* ▲*= good,* ◆*=fair, and* ▼*=poor.*

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of "mixed" type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

木の表現

木 T は階段関数と思えた. つまり, 入力変数の空間の MECE な分割 R_j $(j = 1, \ldots, J)$ と実数 γ_j によって

$$x \in R_j \Rightarrow T(x) = \gamma_j$$

と定められる. より形式的に書くと, $\Theta = \{R_j, \gamma_j\}_{j=1}^{J}$ をパラメータとして,

$$T(x; \Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j)$$

と書ける.

木の最適化

最適なパラメータを見つけたい．普通に考えると，以下を計算したくなる．

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_i)$$

でも，こんなの計算量が多すぎる．どうしたらいいか．最適化の計算を以下のように整理する．

Step.1 R_j を固定して，γ_j を最適化する．大抵簡単な計算．平均とか，最頻値とか．

Step.2 Step.1 の結果を使って，R_j を決定する．大抵難しいので，近似の出番．

Step.2 での近似として良くやるのは，貪欲法．また，損失関数をもっと計算しやすいものに変えることもしばしば．(ようわからん．やってみないと旨味が感じられなそう．)

$$\tilde{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} \tilde{L}(y_i, \gamma_i)$$

$\hat{R}_j = \tilde{R}_j$ として，γ_j の値の決定時には，元の損失関数を使う．

木で Boosting

木で Boosting する場合，m 回目の最適化では以下を最小化する．

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)$$

ただし，f_m は m 回目の最適化の結果で，木の和

$$f_m(x) = \sum_{k=1}^{m} T(x; \Theta_k)$$

とする．先述の通り，この最適化はとても大変．何ならただの木の最適化より大変な場合もある．だけど，楽にできるケースもある．

木で Boosting が楽なケース

- 損失関数が Squared-error の場合.
  各ステップの最適化は, 通常の木の最適化と全く同じ.
- 2 値分類で指数損失関数を使う場合.
  AdaBoost. 特に $\gamma_{jm} \in \{-1, 1\}$ の時, 損失の最小化は, 重み付き誤差率

$$\sum_{i=1}^{N} e^{-y_i f_{m-1}(x_i)} \dot{e}^{-y_i T(x_i;\Theta_m)}$$

  の最小化と等価.
- 一方, 頑健さのために, 損失関数として絶対値誤差, Huber loss, deviance 等を使うと, 頑健さは改善するが, 上記のような単純な計算に落とし込めなくなる.
- 一般論として, 領域の分割方法さえ決まれば, パラメータ γ は比較的容易に求まる.
  領域を決めることは非常に困難で, 良い近似手法を考えることが重要.

最急降下法

関数の最小値を発見する手法．これを応用して木の最適化を考えていく．

- 最も傾斜が急な方向に最も小さくなる位置まで移動する，という処理を繰り返して最小点を探す手法．貪欲法．
- 関数 $f(x_1, \ldots, x_n)$ の最小化を考えるとすると，$\mathbf{x}^{(0)}$ を適当にとって，$m = 1, 2, \ldots$ に対して，収束するまで以下を繰り返す．
  1. 方向を決める．勾配ベクトルの逆方向なので勾配 (gradient) を計算すればよい．

$$g_m = \mathrm{grad} f(\mathbf{x}_i) = \frac{\partial f}{\partial x_i}(\mathbf{x}^{(m)})$$

  2. 距離を決める．

$$\rho_m = \arg\min_{\rho} f(\mathbf{x}^{(m)} - \rho g_m)$$

  3. 値を更新する．

$$\mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} - \rho g_m$$

勾配 Boosting

さっきの話を木の最適化に使いたい．損失関数の合計

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i))$$

の最小化なので，この値を $\mathbf{f} = f(x_i)_{i=1}^{N}$ の関数とみなして，最急降下法を使いたい．が，そのまま適用はできない．問題点は以下．

- 最急降下法では入力値 $\mathbf{x}$ に制約はないが，上記のように定めた $\mathbf{f}$ は高々 J_m 種類の値しか含まない．
- 最急降下法では，向きを完全に決めてから距離を決めるが，木の最適化では，領域の分割を決めてから各領域ごとに独立に高さを計算する．
- 勾配は訓練データに対する値しか考慮していないので，過学習を防げない．

ここら辺の差異を避けて，最急降下法の肝だけを活用する方法として，次の方法が挙げられる．(汎化性能の問題解決できてないのでは...??)

- 領域の決定のために，勾配ベクトルとの残差二乗和を最小化する木を見つける．

$$\hat{\Theta}_m = \arg\min_{\Theta} \sum_{i=1}^{N} (-g_{im} - T(x_i; \Theta))^2$$

- これで定まる各領域 R_{jm} について，良い γ_{jm} の値を計算する．

この 2 段階は，まさしくさっき言ったもの．

勾配 Boosting の実装

- 勾配 Boosting で回帰するアルゴリズム.
- 2-(b) で領域の分割決定.
- 2-(c) で各領域に対応する γ の値を決定.
- 分類についても似たようなもん. 多値分類の場合, 繰り返しの回数は M(クラス数) 回.

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

勾配 Boosting のメタパラメータその 1：木のサイズ J

普通木を作るときの決め方

- 普通の木では, 大きく作って刈り込む作戦がとられたり.
- 刈り込みをする時点で, 「最適な分類」をしようとするので, Boosting で使うと, 不必要に強いものが残りがち.
- 結果, 過剰な計算量と大きすぎる木たち, ということになりがち.

最も単純な回避策

- すべての木のサイズを同じ値 $J_m = J \ \forall m$ にして, J を調整する.
- この J は, モデル自体の変数 (メタパラメータ, ハイパーパラメータ) と思える. (基底関数展開の基底の数や k-NN の k みたいな)

J の値の決め方

- 前に出てきた ANOVA 分解.

$$\eta(X) = \sum_i \eta_i(X_i) + \sum_{i,j} \eta_{i,j}(X_i, X_j) + \cdots$$

- 入力変数の交互作用の程度を考える. 大きさ J の木で起こる交互作用は高々 $J - 1$.
- 交互作用の程度は, 大抵それほど多くない. 経験的に, $J > 10$ が必要なことはほぼない. $4 \leq J \leq 8$ 程度がうまくいく.

勾配 Boosting のメタパラメータその 2：Boosting の回数 M

- M を増やすほど損失は減るが, やりすぎると過学習.
- 丁度良い M の値を見つけなくてはいけない.
- 検証用データを別に用意するのは良くある戦略.
- Neural Network の early stopping とよく似ているとか. (11 章で見る.)

過学習を防ぐ方法として, M の値を操作する以外の方法もある.

過学習を防ぐ方法 1：縮小法

- M ではなく，アルゴリズムをちょっといじって，過学習を防ぐ方法．
- f_m の更新時に係数をかけて，更新幅を縮小する．

$$f_m(x) = f_{m-1} + \nu \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm})$$

- ν が学習率をコントロールしている．小さいほど学習率が低い．
- 経験的に，ν を小さく (≤ 1 とか) とり，M は early stopping することで，いい感じにできる．特に回帰では効果的．
- ただし，計算量は増える．

縮小法の結果

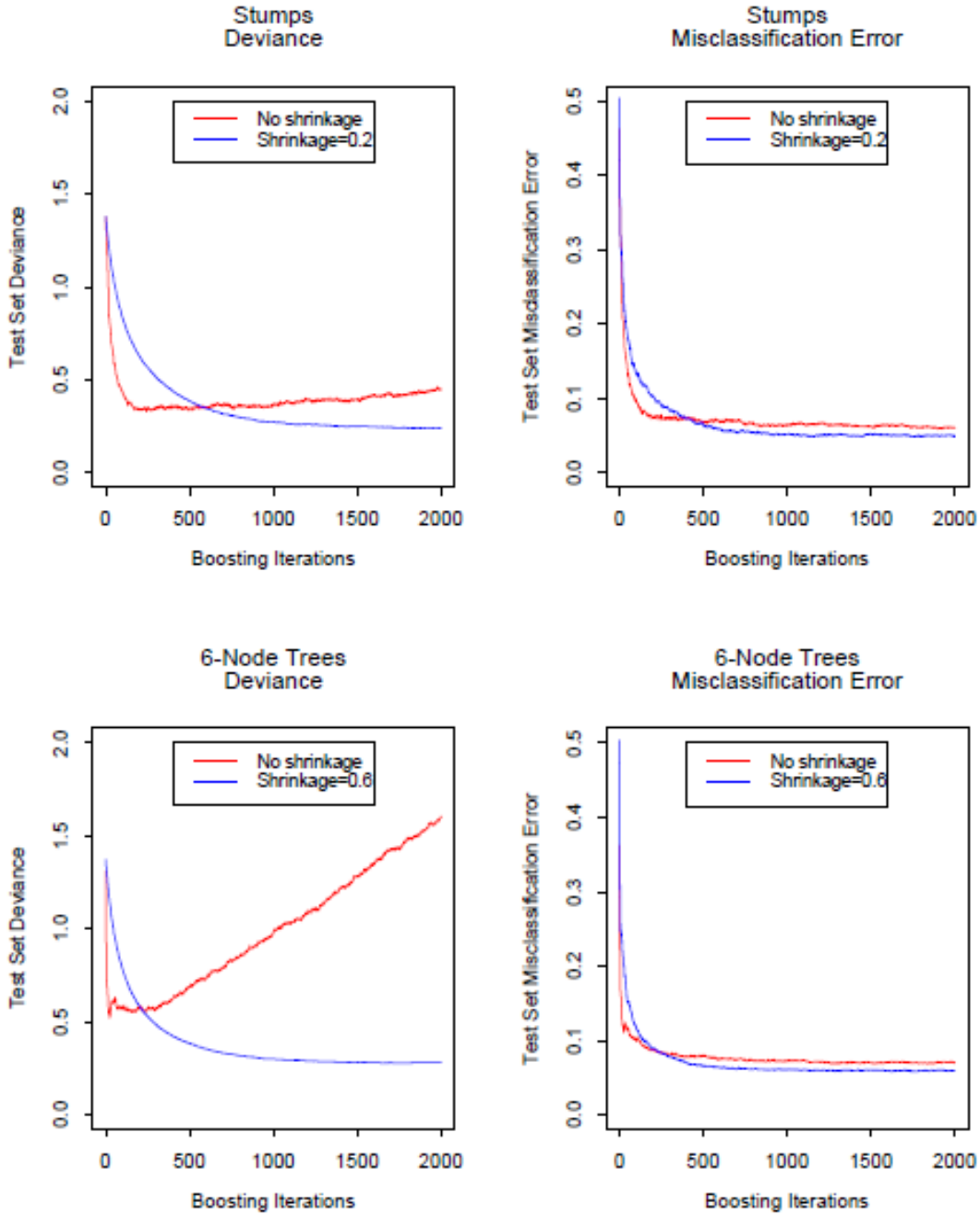


FIGURE 10.11. *Test error curves for simulated example (10.2) of Figure 10.9.*

過学習を防ぐ方法 2：部分標本

- bootstrap 法に似た考え方を使う.
- 繰り返しのたびに, 訓練データから一定数のデータを非復元抽出して, そのデータを使って最適化する.
- 「一定数」は, 訓練データの半数程度が相場. N が大きければもっと少なくてもよい.
- 計算量はもちろん改善. 多くの場合, 精度も良くなる.
- 右図から, 部分標本と縮小法を併用したときが最も良いことが分かる.

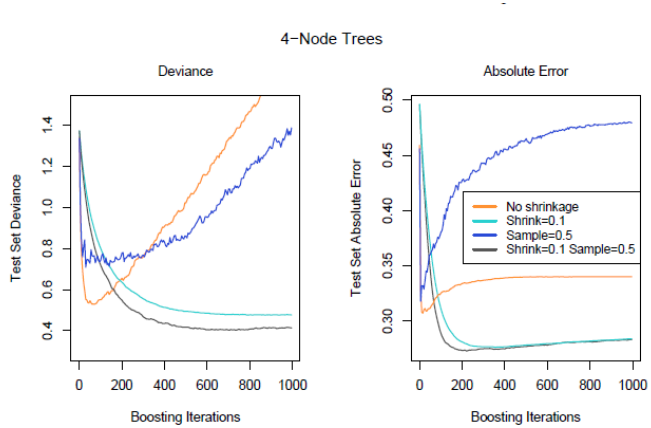


FIGURE 10.12. *Test-error curves for the simulated example (10.2), showing the effect of stochasticity. For the curves labeled "Sample= 0.5", a different 50% subsample of the training data was used each time a tree was grown. In the left panel the models were fit by* `gbm` *using a binomial deviance loss function; in the right-hand panel using square-error loss.*

入力変数の相対的な重要さ

変数の重要さ

- 入力変数のうち, 一部のみが出力変数に大きな影響を与えていることが多い.
- この, “影響度” の大きさを推定することはとても有用.

単体の木における入力変数の重要さ

- 以下の式で表されたり.

$$\mathcal{I}_l^2 = \sum_{t=1}^{J-1} \hat{i}_t^2 I(v(t) = l)$$

- $\hat{i}_t^2$ はその分割における誤差の改善幅.
- これはつまり, 木の中でその変数が分割に使われている部分でどれだけ予測が改善したか, ということ.

木の和への一般化

- 単純に各木における重要さの平均

$$\mathcal{I}_l^2 = \frac{1}{M} \sum_{m=1}^{M} \mathcal{I}_l^2(T_m)$$

- K 値分類の場合, K 個の木の和が作られるので, さらにそれらにおける重要度の平均を取る.

一部の入力変数の集合の重要さ

各変数ごとの重要さの次は, いくつかの入力変数からなる集合の重要さ (部分依存, partial dependence)) を考えたい.

- グラフなどで視覚化すると, 分かりやすい. が, 高次元では難しい.
- Trellis など数次元くらいのデータのうまい可視化の仕方もある. けど限界があるよね.

一部の入力変数の集合の重要さ

- $X_s \subset X$, $X_c = X \backslash X_s$ とする.
- $f(X) = f(X_s, X_c)$ と思える.
- $f_s(X_s) = E_{X_c}[f(X_s, X_c)]$ とすると, 部分依存を表現できそう. 実際, X_s と X_c の間の交互作用が弱いなら, かなりいい.
- 母集団に対する値は難しいので, 以下で推定する.

$$\bar{f}_s(X_s) = \frac{1}{N} \sum_{i=1}^{N} f(X_s, x_{ic})$$

- この計算は一般にはすごく大変だけど, f が木の時は簡単にできる.

部分依存は, 条件付き期待値を使わない

- 条件付き期待値は, X_c の周辺分布を使って平均を計算する.
- 一方, 部分依存関数 $f_s(X_s) = E_{X_c}[f(X_s, X_c)]$ は, 変数 X_c の全域的な分布を使って計算している.
- X_s と X_c が独立なときのみ, 両者は一致する.
- X_s と X_c について, f が完全に加法的なとき, 部分依存関数は X_s部分 $+$ 定数, 乗法的なときは, X_s部分 $\times$ 定数
- 条件付き期待値ではそうきれいにはならない.
- (だから何なのか良く分からん.
- この話も K 値分類に拡張できるよ.