Data Science for Actuaries (ACT6100)

Arthur Charpentier

Aggrégation # 5.2 (Apprentissage séquentiel)

automne 2020

https://github.com/freakonometrics/ACT6100/

Bagging et Boosting

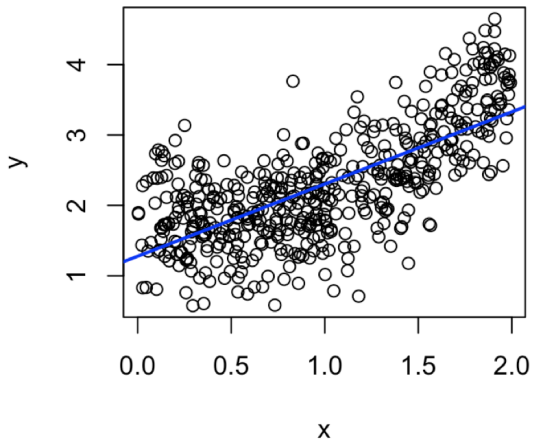
- Le Bagging vise à construire B modèles à partir de B bases de données obtenues par bootstrap. La **construction** de ces B modèles se fait de façon **parallèle**, c'est-à-dire de façon totalement indépendante.
- Le Boosting vise à construire B modèles de façon **séquentielle**: le b-ième modèle dépend du $(b-1)$-ième modèle qui dépend lui-même du $(b-2)$-ième modèle, etc. Chacun nouveau modèle est construit spécifiquement afin d'améliorer les prédictions faites par le modèle précédent.
- En français, on pourrait parler (mais en pratique, on ne le fait jamais) d'amplification ou de stimulation.

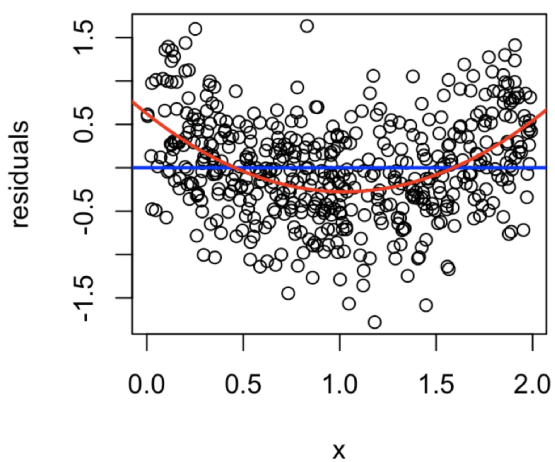
Apprendre de ses erreurs...

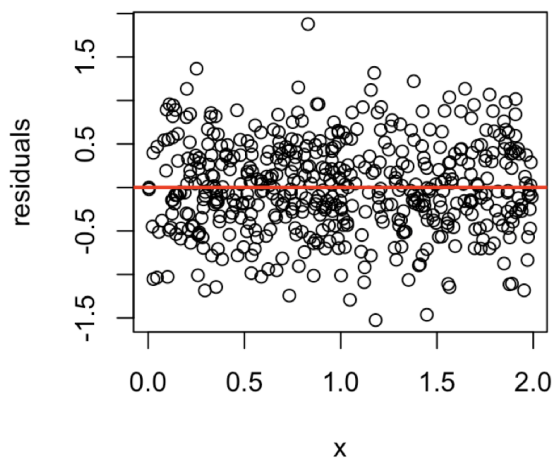
```
> df = dfr = data.frame(x=x, y=y)
> reg1 = lm(y~x, data=df)
> dfr$y = residuals(reg1)
> reg2 = lm(y~poly(x,2), data=dfr)
```

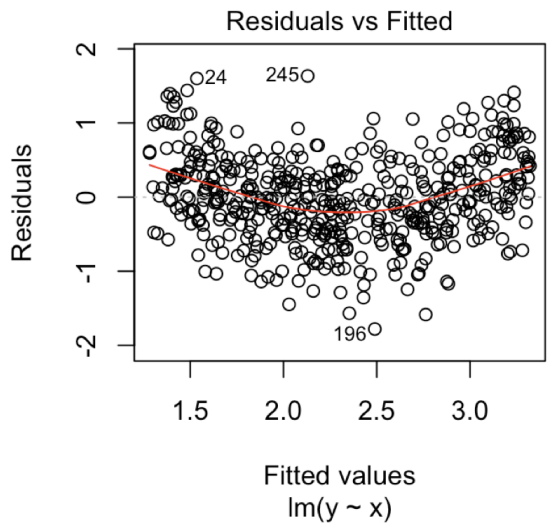
1. $y_i = \beta_0 + \beta_1 x_i + r_{1,i}$
2. $r_{1,i} = \gamma_0 + \gamma_1 x_i + \gamma_2 x_i^2 + r_{2,i}$

(etc ?)



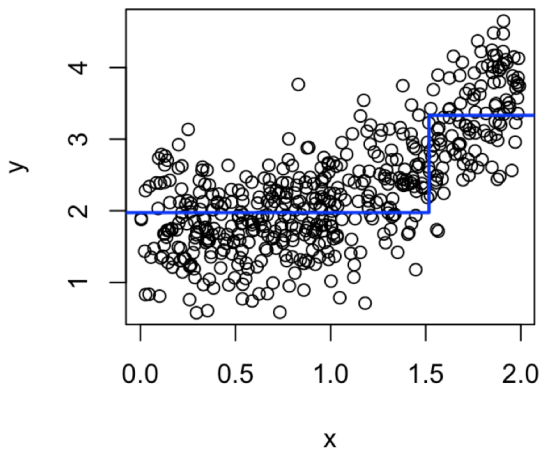


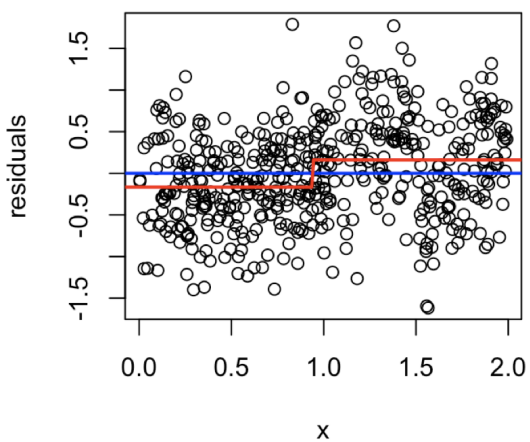


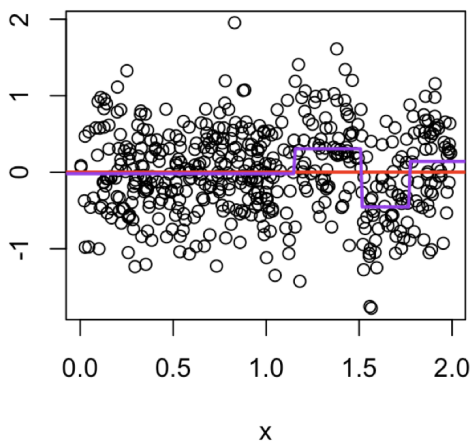


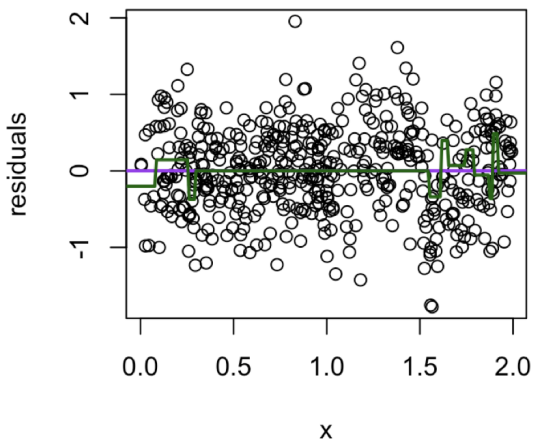
Apprendre de ses erreurs...

```
> df = dfr = data.frame(x=x, y=y)
> reg1 = rpart(y~x, data=df)
> dfr$y = residuals(reg1)
> reg2 = rpart(y~x, data=dfr)
> dfr$y = residuals(reg2)
> reg3 = rpart(y~x, data=dfr)
> dfr$y = residuals(reg3)
> reg4 = rpart(y~x, data=dfr)
```









Boosting

Algorithm 1: Boosting (version 1)

initialization : k (number of trees), γ, $f_0(\boldsymbol{x}) = \overline{y}$;
for $t = 1, 2, ...k$ **do**
    compute $r_{i,t} \leftarrow y_i - f_{t-1}(\boldsymbol{x}_i)$;
    fit a model $r_{i,t} \sim h(\boldsymbol{x}_i)$ for some tree h;
    update $f_t(\cdot) = f_{t-1}(\cdot) + \gamma h(\cdot)$

Algorithm 2: Boosting (version 2)

initialization : k (number of trees), $f_0(\boldsymbol{x}) = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} \ell(y_i, \gamma)$;
for $t = 1, 2, ...k$ **do**
    compute $r_{i,t} \leftarrow \left. \frac{\partial \ell(y_i, \widehat{y})}{\partial \widehat{y}} \right|_{\widehat{y}=f_{t-1}(\boldsymbol{x}_i)}$;
    fit a model $r_{i,t} \sim h(\boldsymbol{x}_i)$ for some tree h;
    update $f_t(\cdot) = f_{t-1}(\cdot) + \gamma h(\cdot)$

Algorithme AdaBoost

ou voir la mise à jour à l'aide de poids (si la classification n'est pas bonne)

On s'intéresse à un problème de classification binaire (0 ou 1) à partir d'une base de données $\mathcal{D} = \{Y_i; \mathbf{X}_i\}_{i=1,\dots,n}$.

1. Toutes les observations reçoivent un poids identique: $p_1(\mathbf{X}_i) = 1/n,\ i = 1, \dots, n.$

Algorithme AdaBoost (suite)

2. Pour l'étape $t = 1, \ldots, T$,
   - 2a. Piger au hasard (en utilisant les poids $p_t(\mathbf{X}_i)$) une base de données d'entrainement $\mathcal{D}_t = (\mathcal{D}, p_t)$.
   - 2b. Apprendre une règle de classification r_t sur cette base de données.
   - 2c. Calculer l'erreur apparente

$$\epsilon_t = p_t\left(r_t(\mathbf{X}_i) \neq Y_i\right) = \sum_{i: r_t(\mathbf{X}_i) \neq Y_i} p_t(i)$$

     pour $\mathcal{D}_t$ et évaluer $\alpha_t = 0.5 \ln((1 - \epsilon_t)/\epsilon_t)$.
   - 2d. Mettre à jour les poids:

$$p_{t+1}(\mathbf{X}_i) = \begin{cases} \frac{p_t(\mathbf{X}_i)}{Z_t} e^{-\alpha_t}, & r_t(\mathbf{X}_i) = Y_i \\ \frac{p_t(\mathbf{X}_i)}{Z_t} e^{+\alpha_t}, & r_t(\mathbf{X}_i) \neq Y_i, \end{cases}$$

     où Z_t est une constante de normalisation assurant que $\sum_{i=1}^{n} p_{t+1}(\mathbf{X}_i) = 1$.

Algorithme AdaBoost (suite)

3. La classification finale faite par le modèle est alors

$$R(\mathbf{X}_i) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t r_t(\mathbf{X}_i) > 0 \\ -1, & \sum_{t=1}^{T} \alpha_t r_t(\mathbf{X}_i) < 0. \end{cases}$$

Algorithm 3: Boosting (Adaboost)

initialization : k, $\omega_i \leftarrow 1/n$;
for $t = 1, 2, \ldots k$ **do**
  error rate $e_t \leftarrow \dfrac{\sum_{i=1}^{n} \omega_i \mathbf{1}(y_i \neq f_{t-1}(\boldsymbol{x}_i))}{\sum_{i=1}^{n} \omega_i}$;
  set $\alpha_t \leftarrow \log(1 - e_t) - \log(e_t)$;
  update $\omega_i \leftarrow \omega_i \; e^{\alpha_t \mathbf{1}(y_i \neq f_{t-1}(\boldsymbol{x}_i))}$;
  update $f_t(\cdot) \leftarrow f_{t-1}(\cdot) + \alpha_t h(\cdot)$

Remarques

On apprend tant que $\alpha_t \geq 0$, soit $e_t < 1/5$.
Modèle classique pour h: CART (avec des poids)

- Approche théorique

Étant donné une famille de modèles $\mathcal{H}$, on cherche à résoudre

$$h^\star = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \left\{ \mathbb{E}\big[\ell(Y, h(\boldsymbol{X}))\big] \right\}$$

ou sa version empirique

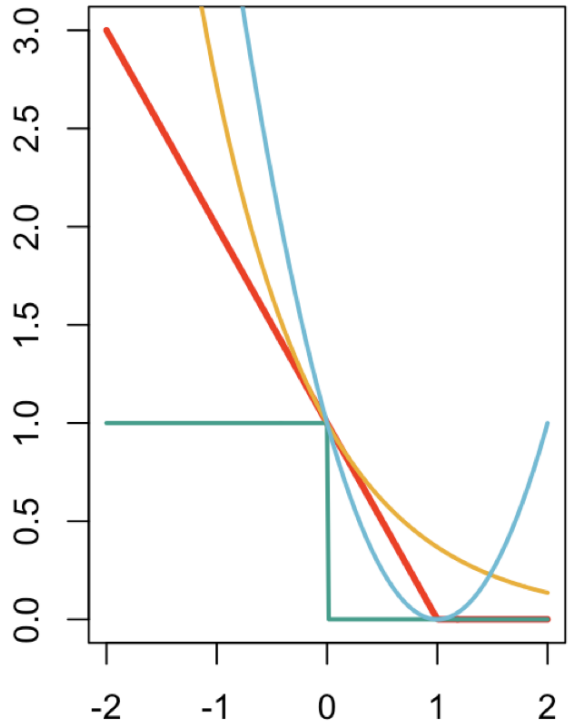
$$\widehat{h}^\star = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(\boldsymbol{x}_i)) \right\}$$

avec classiquement $\ell(y, \hat{y}) = \mathbf{1}(y = \hat{y})$.
On peut tenter de convexifier la fonction de perte,
$\overline{\ell}(y, \hat{y}) = \exp(-y\hat{y})$.

Remarques

Supposons $y \in \{-1, +1\}$.
Les points sont mal classés si $y \cdot f(\boldsymbol{x} < 0)$, on peut tracer la fonction de perte en fonction de $y\hat{y}$

- misclassification: $\mathbf{1}(y\hat{y} < 0)$
- hinge: $(1 - y\hat{y})_+$
- exponential: $\exp(-y\hat{y})$



L'algorithme Adaboost vérifie $f_t = f_{t-1} + 2\beta^\star h^\star$, où $(\beta^\star, h^\star)$ est solution de

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n} \overline{\ell}\big(y_i, f_{t-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i)\big) \right\}$$

Remarques

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\text{argmin}} \left\{ \sum_{i=1}^{n} \overline{\ell}\big(y_i, f_{t-1}(\boldsymbol{x}_i) + \beta h(\boldsymbol{x}_i)\big) \right\}$$

$$(\beta^\star, h^\star) = \underset{(\beta,h)}{\text{argmin}} \left\{ \sum_{i=1}^{n} \underbrace{\exp\big(-y_i f_{t-1}(\boldsymbol{x}_i)\big)}_{\omega_i} \exp\big(-y_i \beta h(\boldsymbol{x}_i)\big) \right\}$$

on distingue alors les i pour lesquels $y_i = h(\boldsymbol{x}_i)$ et $y_i \neq h(\boldsymbol{x}_i)$, et

$$h^\star = \underset{h \in \mathcal{H}}{\text{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \omega_i \ell(y_i, h(\boldsymbol{x}_i)) \right\}$$

Remarques

- L'erreur apparente est calculée en utilisant la distribution avec laquelle le modèle est entrainé.
- Chacune des nouvelles règles r_t est pondérée par un poids α_t qui indique l'importance que cette règle aura dans la décision finale R.
- Si T, le nombre de modèles, est *trop* grand, alors il y a un risque que l'algorithme se concentre trop vers la fin sur les cas difficiles (potentiellement du bruit) et accorde à ces derniers trop d'importance dans la décision finale R.

Bornes

- L'erreur apparente

$$\epsilon_t = p_t\left(r_t(\mathbf{X}_i) \neq Y_i\right) = \sum_{i:r_t(\mathbf{X}_i)\neq Y_i} p_t(i)$$

peut être réécrite comme

$$\epsilon_t = 0.5 - \gamma_t,$$

où γ_t est l'amélioration apportée à la prédiction par la règle r_t en comparaison avec le hasard (0.5).

Bornes (suite)

- On peut démontrer que l'erreur apparente finale (pour T) a comme borne supérieure

$$\epsilon_T \leq \exp\left(-2\sum_t \gamma_t^2\right) \leq \exp\left(-2T\gamma^2\right),$$

où $\gamma = \min(\gamma_1, \ldots, \gamma_T)$.

- Ainsi, si le weak learner fait légèrement mieux que le hasard, $\gamma_t > 0$ et l'erreur apparente finale diminue exponetiellement avec le nombre de modèles (T).

Bornes (suite)

- On peut également démontrer que l'erreur de généralisation, c'est-à-dire l'erreur calculée sur une base de données qui n'a pas été utilisée pour l'entrainement du modèle, a comme borne supérieure

$$\text{Err}_G \leq \epsilon_T + \sqrt{\frac{(T)(d_{\mathcal{H}})}{n}},$$

où $d_{\mathcal{H}}$ est un terme lié à la taille de l'espace d'hypothèses.

- Cette borne indique que si T devient grand, alors le Boosting devrait tendre à surapprendre.
- En pratique, on observe rarement ce phénomène.

Fonction objectif

1. De façon générale, on cherche une règle $\widehat{r}$ en résolvant un problème d'optimisation empirique

$$\widehat{r} = \underset{r \in \mathcal{R}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, r(\boldsymbol{X}_i)),$$

   où ℓ est une fonction de perte quelconque.

2. Si l'optimisation est faite sur l'ensemble des fonctions r possibles et imaginables, le problème est trop complexe pour être résolu (même numériquement!).

Fonction objectif

- Une solution possible est de convexifier la fonction objectif afin de rendre le problème plus simple d'un point de vue numérique.
- L'algorithme AdaBoost répond à ce principe de minimisation pour un risque convexifié donné par

$$\ell(Y_i, r(\boldsymbol{X}_i)) = \exp\left(-Y_i r(\boldsymbol{X}_i)\right).$$

- On va également réaliser l'optimisation de façon séquentielle plutôt que globale (greedy).

Fonction objectif

On peut démontrer que la règle de classification à l'étape t de l'agorithme AdaBoost peut s'écrire comme étant

$$r_t(\mathbf{X_i}) = r_{t-1}(\mathbf{X}_i) + \alpha_t R_t(\mathbf{X}_i),$$

où

$$(\alpha_t, R_t) = \underset{(\alpha, r) \in \mathbb{R} \times \mathcal{R}}{\operatorname{argmin}} \sum_{i=1}^{n} e^{-Y_i (r_{t-1}(\mathbf{X}_i) + \alpha r(\mathbf{X}_i))}$$

et où $\mathcal{R}$ est l'espace des choix possibles pour la règle r.

Boosting par descente de gradient fonctionnelle

1. Initialiser

$$r_0(\mathbf{X}_i) = \underset{c \in \mathbb{R}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, c).$$

2. Pour les étapes $t = 1, \ldots, T$,
   2a. Calculer

$$U_i = - \left[\frac{\partial L(Y_i, r(\mathbf{X}_i))}{\partial r(\mathbf{X}_i)} \right] |_{\{r(\mathbf{X}_i) = r_{t-1}(\mathbf{X}_i)\}}, \qquad i = 1, \ldots, n.$$

   2b. Ajuster le weak learner sélectionné sur l'échantillon composé des éléments $(\mathbf{X}_1, U_1), (\mathbf{X}_2, U_2), \ldots, (\mathbf{X}_n, U_n)$ afin d'obtenir la pseudo-règle $h_t(\mathbf{X}_i)$.
   2c. Effectuer la mise à jour $r_t(\mathbf{X}_i) = r_{t-1}(\mathbf{X}_i) + \alpha h_t(\mathbf{X}_i)$.
3. La prédiction finale est donnée par $r_T(\mathbf{X}_i)$.

Remarques

- Le choix du paramètre α (taux d'apprentissage) est peu important. On recommande généralement de prendre une petite valeur (0.01 ou 0.001).
- Si on pose $\alpha = 1$ et $\ell(y, \hat{y}) = \exp(-y\hat{y})$, on retrouve (presque) l'algorithme AdaBoost.

Gradient Boosting with R, ℓ_2 loss

```
> loss_L2 = function(y, yhat) (mean(1/2*(y-yhat)^2))
> gradient_L2 = function(y, yhat) {(y-yhat)}
```

then use

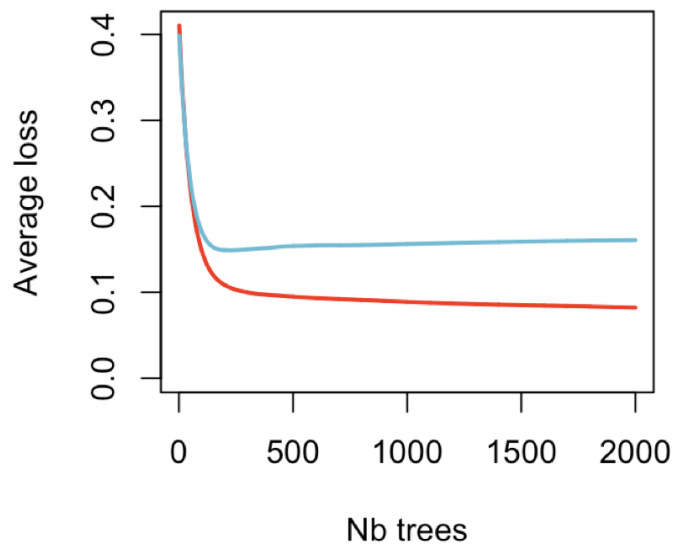
```
> grad_boost_train_valid <- function(formula,
    data_train, data_valid, nu = 0.01, stop=1000, =
    loss_L2) {
  data = as.data.frame(data_train)
  formula = terms.formula(formula)
  noms_X = names(data_train)[-which(names(data_train)
    ==as.character(formula)[2])]
  noms_Y = names(data_train)[which(names(data_train)==
    as.character(formula)[2])]
  y = data_train[, noms_Y]
  fit_train = fit_valid = mean(y)
  data = as.data.frame(data_train[,noms_X])
  names(data) = noms_X
  data$u = grad.fun(y = y, yhat = fit_train)
  loss_train = loss_valid = c()
```

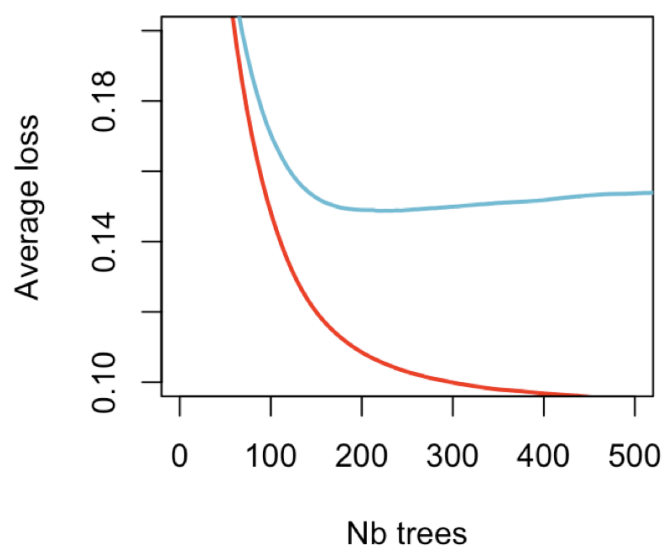
Gradient Boosting with R, ℓ_2 loss

```
  for (i in 1:stop) {
    model_weak = rpart(u  ~ ., data=data, maxdepth=3,
    cp=1e-9)
    fit_train = as.numeric(fit_train + nu * predict(
    model_weak, newdata=data_train))
    fit_valid = fit_valid + nu * predict(model_weak,
    newdata=data_valid)
    data$u = as.numeric(unlist(grad.fun(y = as.numeric
    (unlist(data_train[noms_Y])), yhat = fit_train)))
    loss_train <- c(loss_train, loss.fun(y = as.
    numeric(unlist(data_train[noms_Y])), yhat =
    fit_train))
    loss_valid <- c(loss_valid, loss.fun(y = as.
    numeric(unlist(data_valid[noms_Y])), yhat =
    fit_valid))
  }
  loss_matrix = data.frame(N = 1:stop, TRAIN =
    loss_train, VALID = loss_valid)
  return(list(loss = loss_matrix))
}
```

Gradient Boosting with R, ℓ_2 loss

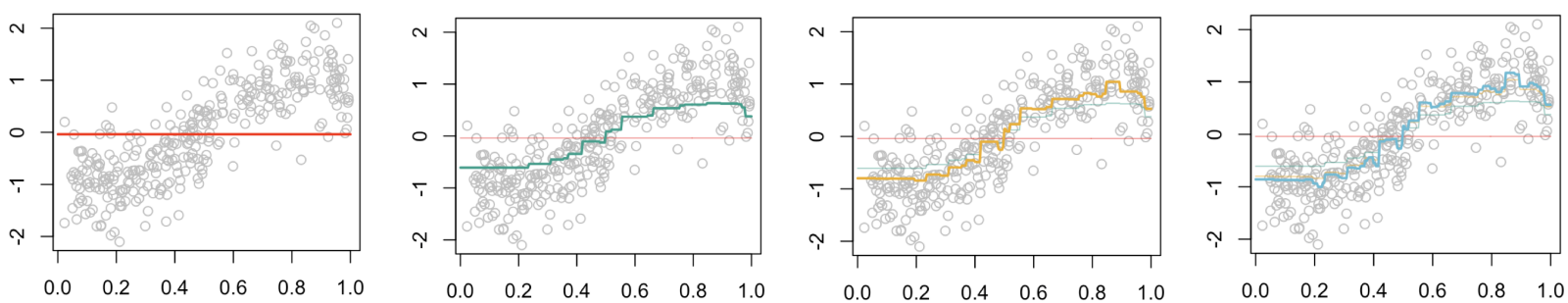
```
> x=sort(runif(n))
> y=sin(-3*pi/4+x*pi*3/2)+rnorm(n)/2
DF = data.frame(x=x,y=y)
> set.seed(1)
> idx = sample(1:n,size = 2*n/3)
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,],
    data_valid=DF[-idx,], nu = .01, stop=2000,grad.fun
     = gradient_L2, loss.fun = loss_L2)
```





Gradient Boosting with R, ℓ_2 loss

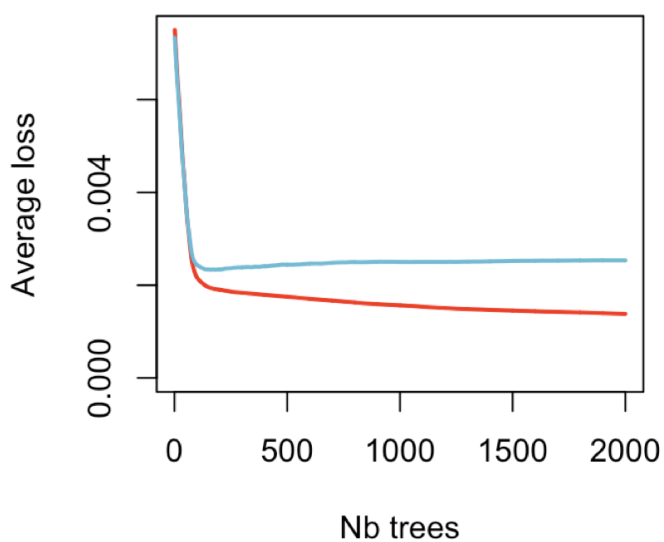
```
> x=sort(runif(n))
> y=sin(-3*pi/4+x*pi*3/2)+rnorm(n)/2
DF = data.frame(x=x,y=y)
> set.seed(1)
> idx = sample(1:n,size = 2*n/3)
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,],
    data_valid=DF[-idx,], nu = .01, stop=2000,grad.fun
     = gradient_L2, loss.fun = loss_L2)
```

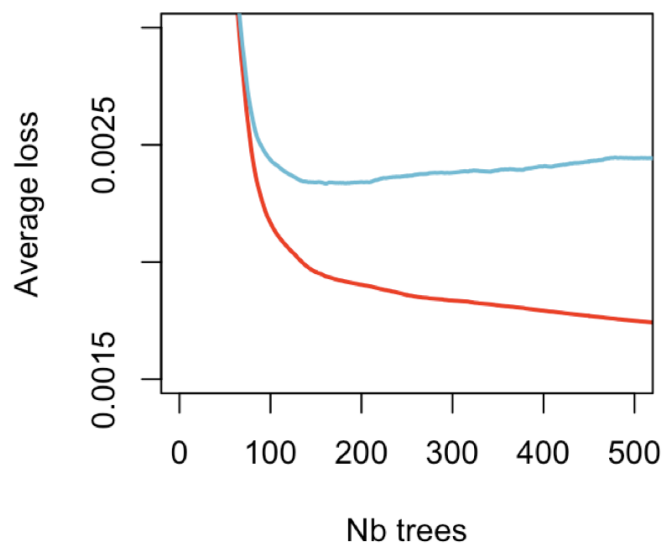


Sequential learning, with 0, 100, 200, 500 trees

Gradient Boosting with R, ℓ_1 (or Huber) loss

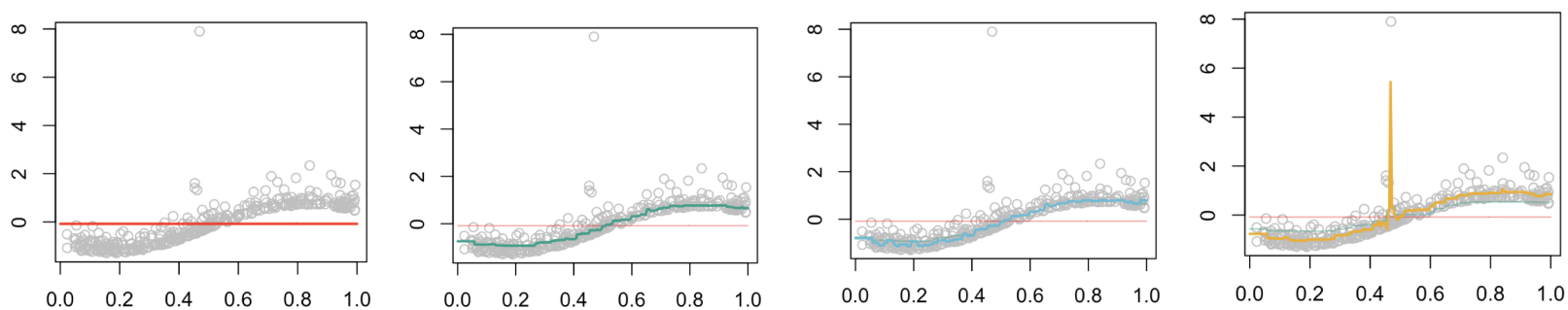
```
> loss_L_huber = function(y, yhat, delta=.01){
  L=1/2*(y-yhat)^2*(abs(y-yhat)<=delta) + delta*(abs(y
    -yhat)-delta/2)*(abs(y-yhat)>delta)
  return(mean(L))}
> gradient_L_huber = function(y, yhat, delta=.01){
  L=(y-yhat)*(abs(y-yhat)<=delta) + delta*sign(y-yhat)
    *(abs(y-yhat)>delta)
  return(as.numeric(L))}
> y = sin(-3*pi/4+x*pi*3/2)
> e = rlnorm(n)
> DF = data.frame(x=x,y=y+(e-mean(e))/sd(e)/2)
```





Gradient Boosting with R, ℓ_1 (or Huber) loss

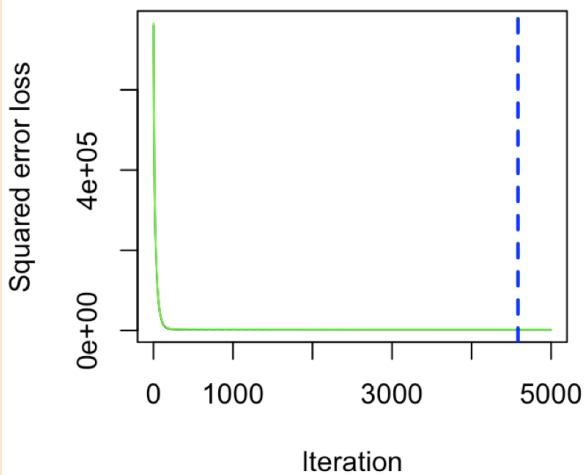
```
> library(rpart)
R = grad_boost_train_valid(y~x, data_train=DF[idx,],
    data_valid=DF[-idx,], nu = .01, stop=2000,grad.fun
     = gradient_L_huber, loss.fun = loss_L_huber)
```



Sequential learning, with 0, 100, 1000 trees,
(and 1000 trees with some ℓ_2 loss on the right)

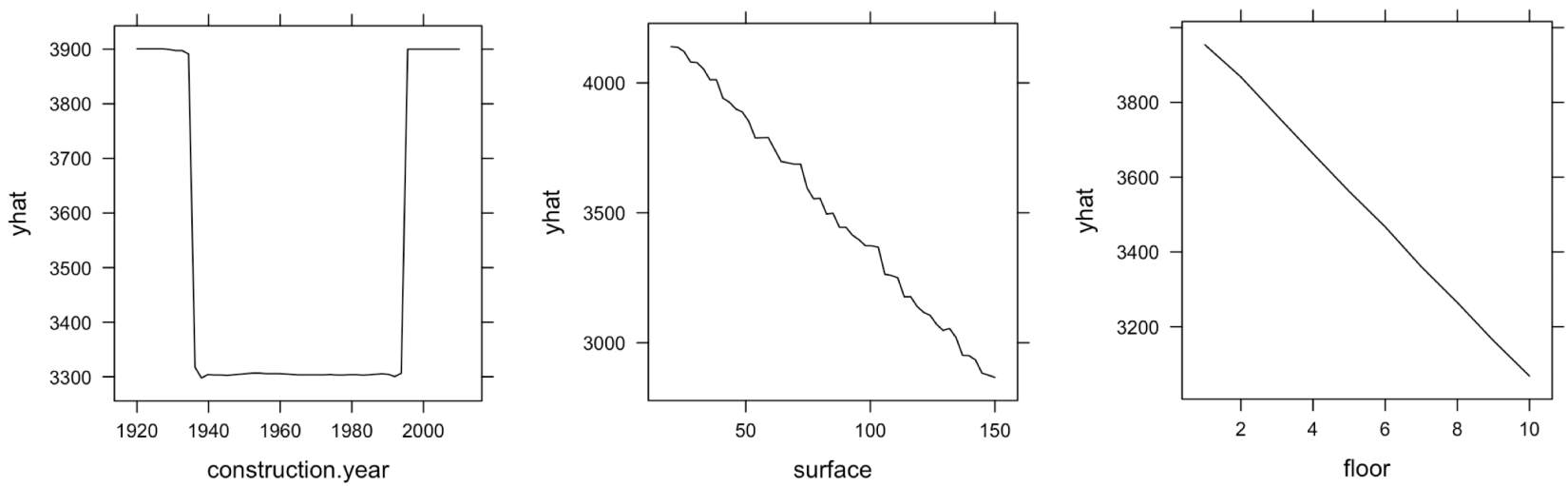
Gradient Boosting with R, ℓ_2 loss

```
> library(gbm)
> library(DALEX)
> G = gbm(
  formula = m2.price ~ .,
  data = apartments,
  distribution = "gaussian",
  n.trees = 5000,
  shrinkage = 0.1,
  cv.folds = 5
)
> gbm.perf(G, method = "cv")
> min_MSE <- which.min(G$cv.error)
> sqrt(G$cv.error[min_MSE])
[1] 50.67178
```



Gradient Boosting with R, ℓ_2 loss

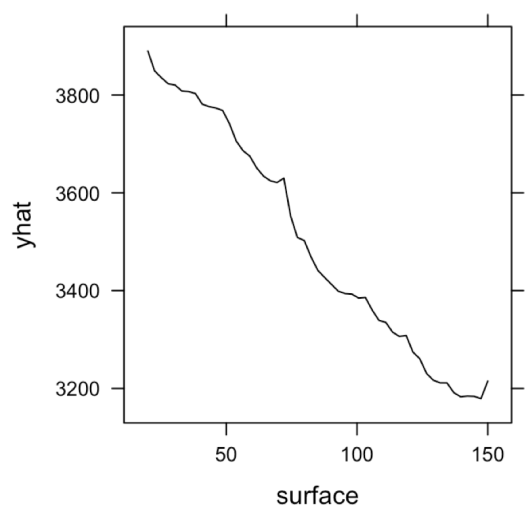
```
> library(pdp)
> pdp::partial(G, pred.var = "floor", plot = TRUE,n.
    trees=2000)
```

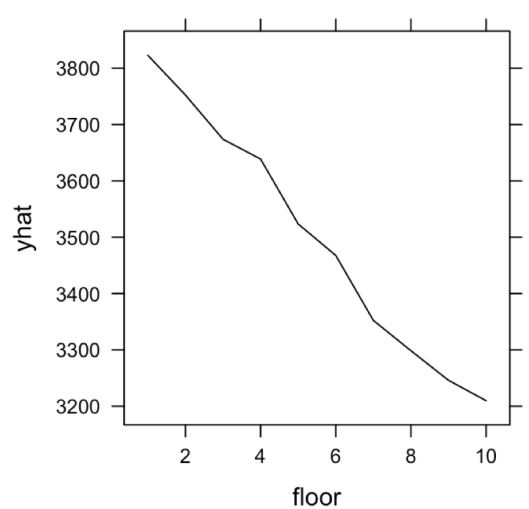


Gradient Boosting with R, ℓ_2 loss

one can compare with a random forest

```
> library(randomForest)
> RF = randomForest(m2.price ~ ., data = apartments)
> pdp::partial(G, pred.var = "floor", plot = TRUE,n.
    trees=2000)
```





GLM & boosting

Consider a logistic regression, with data $(y, \boldsymbol{x})$ and

$$\mathbb{P}(Y = y | \boldsymbol{X} = \boldsymbol{x}) = p(\boldsymbol{x})^y \big[1 - p(\boldsymbol{x})\big]^{1-y},$$

with

$$p(\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^\top \boldsymbol{\beta}}}{1 + e^{\boldsymbol{x}^\top \boldsymbol{\beta}}} = \frac{1}{1 + e^{-\boldsymbol{x}^\top \boldsymbol{\beta}}}$$

Here, given $\psi : \mathbb{R}^p \to \mathbb{R}$, consider

$$p(\boldsymbol{x}) = \frac{e^{\psi(\boldsymbol{x})}}{e^{-\psi(\boldsymbol{x})} + e^{\psi(\boldsymbol{x})}} = \frac{1}{1 + e^{-2\psi(\boldsymbol{x})}}$$

i.e.

$$\psi(\boldsymbol{x}) = \frac{1}{2} \log \left(\frac{p(\boldsymbol{x})}{1 - p(\boldsymbol{x})} \right)$$

We want to find $\psi^\star$ that solves $\psi^\star = \text{argmin}\big\{ -\log \mathcal{L} \big\}$, i.e

$$\psi^\star = \text{argmin} \left\{ \frac{1}{n} \sum_{i=1}^{n} \log \big[1 + \exp[-2\tilde{y}_i \psi(\boldsymbol{x}_i)]\big] \right\}$$

where $\tilde{y}_i = 2y_i - 1 \in \{-1, +1\}$.

Poisson Regression & boosting

Recall that $Y \sim \mathcal{P}(\lambda)$ if $\mathbb{P}[Y = y] = \dfrac{e^{-\lambda}\lambda^y}{y!}$

In the Poisson regression, with a log-link function

$$\mathbb{P}[Y = y|\boldsymbol{X} = \boldsymbol{x}] = \frac{e^{-\lambda_{\boldsymbol{x}}}\lambda_{\boldsymbol{x}}^y}{y!}, \quad \text{with } \lambda_{\boldsymbol{x}} = e^{\boldsymbol{x}^\top \boldsymbol{\beta}}$$

The log-likelihood for a sample $(y_i, \boldsymbol{x}_i)$, $i = 1, \cdots, n$ is

$$\log \mathcal{L} = \sum_{i=1}^{n} - \exp\left[\boldsymbol{x}_i^\top \boldsymbol{\beta}\right] + y_i \boldsymbol{x}_i^\top \boldsymbol{\beta} - log y_i!$$

As previously, assume that $\lambda = \exp[\psi(\boldsymbol{x})]$ so that

$$\log \mathcal{L} = \sum_{i=1}^{n} - \exp\left[\psi(\boldsymbol{x}_i)\right] + y_i \psi(\boldsymbol{x}_i) - log y_i! = -\sum_{i=1}^{n} \ell(y_i, \lambda_i)$$

Poisson Regression & boosting

then

$$-\frac{\partial \ell(y_i, \lambda_i)}{\partial \psi(\boldsymbol{x}_i)} = -\exp\left[\psi(\boldsymbol{x}_i)\right] + y_i$$

Let $r_i = y_i - \exp\left[\psi(\boldsymbol{x}_i)\right]$, we want to find the step direction that best fits $-\boldsymbol{r}$, in the least-square sense,

$$h^\star = \text{argmin} \left\{ \sum_{i=1}^{n} \left[-r_i - h(\boldsymbol{x}_i) \right]^2 \right\}$$

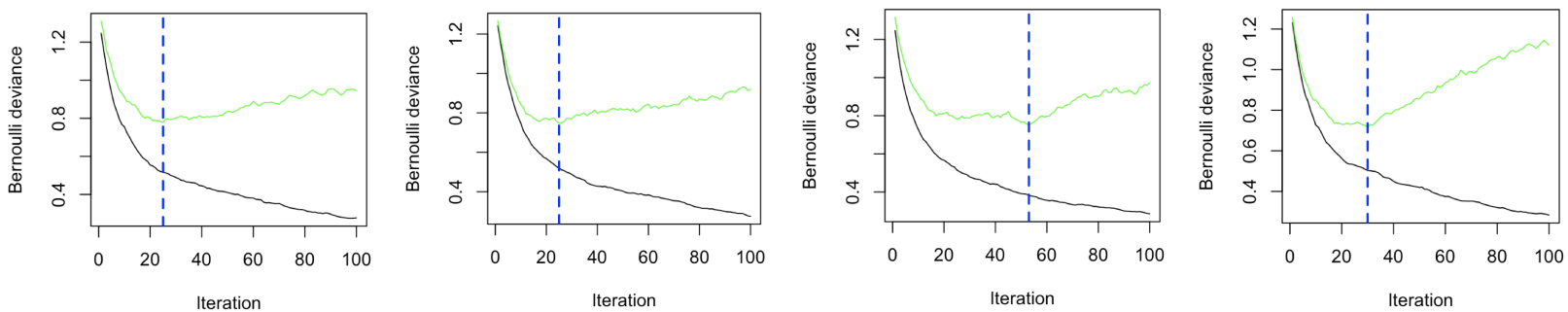
$$h^\star = \text{argmin} \left\{ \sum_{i=1}^{n} \left[\exp\left[\psi(\boldsymbol{x}_i)\right] - y_i - h(\boldsymbol{x}_i) \right]^2 \right\}$$

e.g. with a regression tree

Gradient Boosting with R, logistic loss

one can compare with a random forest

```
> G <- gbm(formula = (PRONO=="SURVIE") ~ ., data =
    myocarde, distribution = "bernoulli", n.trees =
    100, shrinkage = 0.1, cv.folds = 5)
> gbm.perf(G, method = "cv")
```




```
> G
gbm(formula = (PRONO == "SURVIE") ~ ., distribution =
    "bernoulli")
A gradient boosted model with bernoulli loss function.
100 iterations were performed.
The best cross-validation iteration was 30.
There were 7 predictors of which 7 had non-zero
    influence.
```

Gradient Boosting with R, logistic loss

```
> summary(G)
> pdp::partial(G, pred.var = "INSYS", plot = TRUE,n.
    trees=25)
```