Lecture 7: Boosting

Rebecka Jörnsten, Mathematical Sciences

MSA220/MVE441 Statistical Learning for Big Data

3-4th April 2025

Ensemble Methods

Ensemble Methods

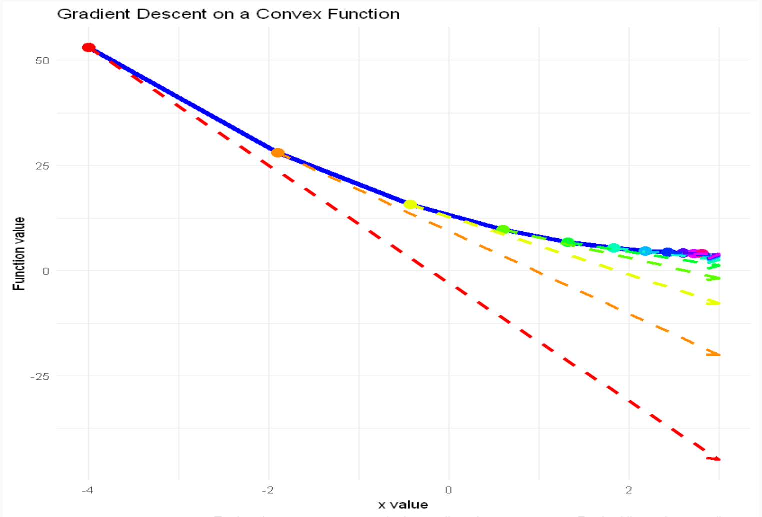
- Bias-Variance trade-off is key to method performance on future data
- If a method is biased we will make systematic errors when predicting on future data
- If our method has high estimation variance we are likely to overfit to the training data and thus generalize poorly - think wiggly decision boundaries or creating isolated regions in order to predict outliers in the training data.
- Bagging: We generate many version of the same model from samples of training data. If each method suffers from overfitting, the randomness of the sampling of multiple training data will cancel out these noisy predictions.
- RandomForest: apply bagging to deep tree models with additional randomness in the allowed data splits as you grow your tree.

Ensemble Methods

- Boosting approaches the problem from a different standpoint
- What if we can learn the model step-by-step and avoid the high estimation variance by stopping before overfitting takes place?
- The idea is that we will learn a sequence of ”weak learners” that on their own suffers from bias (and low estimation) but by weighing them together in an optimal fashion, bias is reduced.
- Note - the difference to bagging and RF is that the contributions of the different methods are *optimally weighted* as opposed to a majority vote.

Gradient descent - a recap

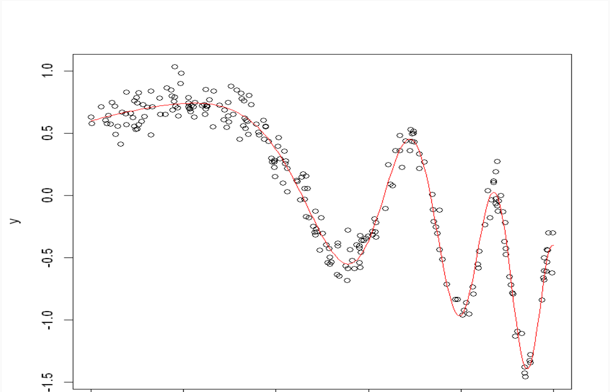
Gradient descent



Each color represents a step in the gradient descent process. Dashed lines show gradients.

Functional gradient descent

In class I talked about the infinite-dimensional parameter estimation problem of estimating functions.
Let us denote the true (univariate) function $f(x)$ and assume we observe this function with additive noise; y.



Functional gradient descent

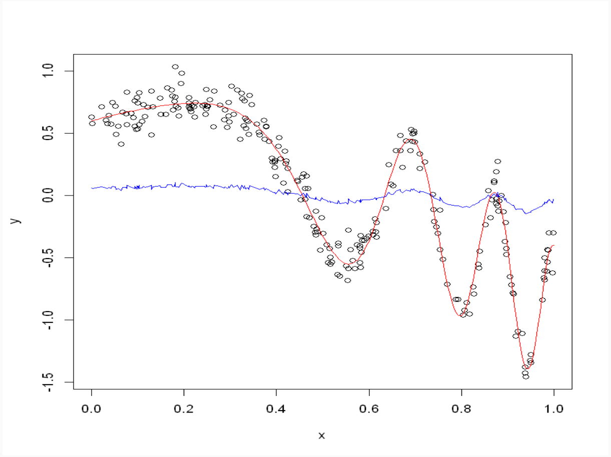
If we use the L2 loss and try estimate the function f from data with vanilla gradient descent (vanilla meaning I don't use optimal or adaptive step sizes), here are the steps;

1. Initialize with $f_t(x_i) = 0, \forall i, t = 0$
2. Compute the pointwise gradient w.r.t. f from the L2 = $\sum_i (y_i - f(x_i))^2$; $\nabla_f L = -(y_i + f_t(x_i))$
3. Update $f_{t+1}(x_i) = f_t(x_i) + \eta\,(y_i - f_t(x_i))$ (sign change due to negative gradient update)

The learning rate η is crucial. If you pick η too small, convergence can be slow. η too large can lead to divergence.

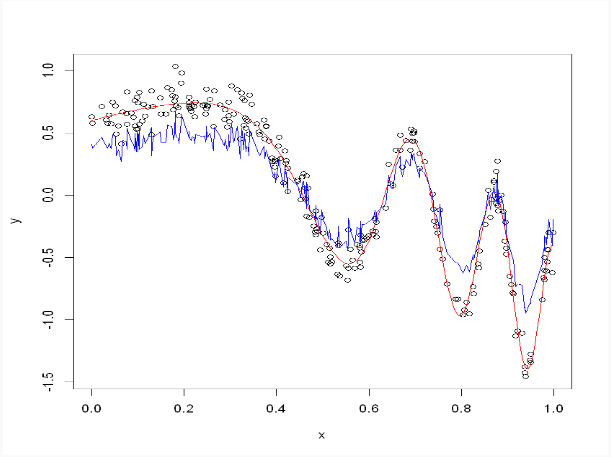
Functional gradient descent

We iterate these steps to approximate the function better and better: step 1



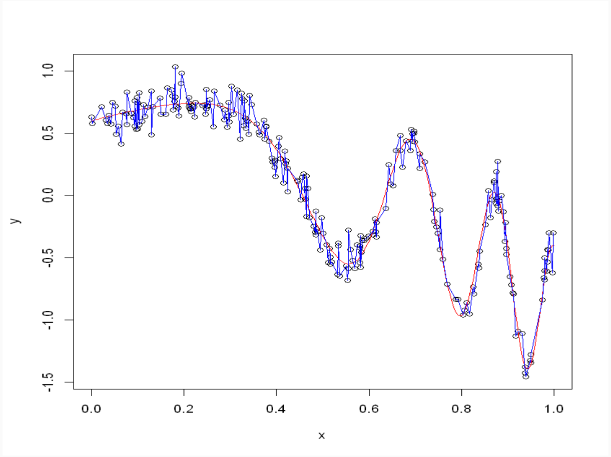
Functional gradient descent

We iterate these steps to approximate the function better and better: step 10



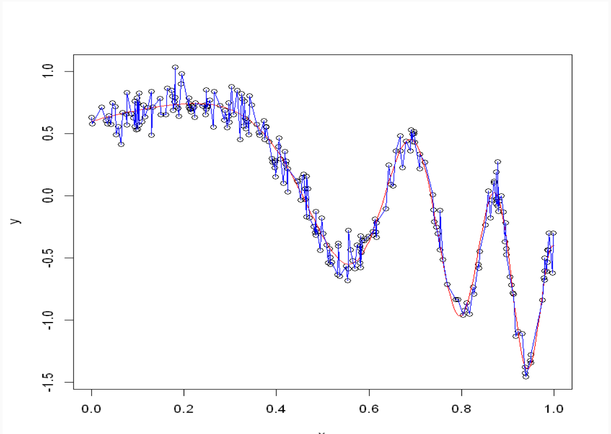
Functional gradient descent

We iterate these steps to approximate the function better and better: step 100



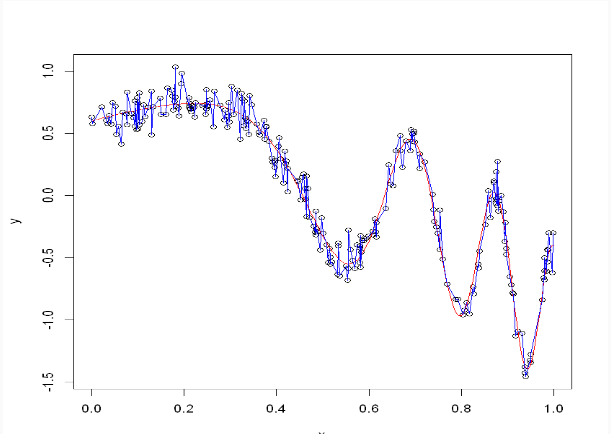
Functional gradient descent

After a few iterations we are able to approximate the function at each training data point perfectly, but we are not approaching the true function due to overfitting.



Functional gradient descent

We have two problems with the current setup; (1) it doesn't allow us to predict outside the training set because we haven't formulated a model and (2) we can't control the overfitting since we are focused on pointwise estimation.



Functional gradient descent

In non-parametric statistics we handle these problems by assuming that the true function f is smooth and use for example regression splines with constraints on the smoothness to achieve a better fit (e.g. constraints on the derivatives of f).

We will focus on the kernel formulation from last lecture here.

Kernel Ridge Regression

KRR

How do we control the smoothness of the function f? The idea is that you can approximate the function well from neighboring locations.

KRR: from the previous lecture on kernel PCA and an kRR we saw that we could arrive at alternative formulation of the ridge regression estimates.

The variables $\widehat{\beta}$ were called the **primal variables** and we defined the **dual variables**

$$\widehat{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_n)^{-1}\mathbf{y} \quad \Rightarrow \quad \widehat{\beta} = \mathbf{X}^\top \widehat{\alpha} = \sum_{l=1}^{n} \widehat{\alpha}^{(l)} \mathbf{x}_l.$$

KRR

Using the dual variables, computed with a chosen kernel, as weights for the observations to compute the primal variables is called **kernel ridge regression**

Standard ridge regression is recovered when using the linear kernel

$$k(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y}.$$

In normal ridge ression, we predict for unseen test data $\mathbf{x}$ as

$$\widehat{f}(\mathbf{x}) = \widehat{\boldsymbol{\beta}}^\top \mathbf{x} = \sum_{l=1}^{n} \widehat{\alpha}^{(l)} \mathbf{x}_l^\top \mathbf{x}$$

KRR

Using the dual variables, computed with a chosen kernel, as weights for the observations to compute the primal variables is called **kernel ridge regression**

Standard ridge regression is recovered when using the linear kernel

$$k(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y}.$$

In normal ridge ression, we predict for unseen test data $\mathbf{x}$ as

$$\widehat{f}(\mathbf{x}) = \widehat{\boldsymbol{\beta}}^\top \mathbf{x} = \sum_{l=1}^{n} \widehat{\alpha}^{(l)} \mathbf{x}_l^\top \mathbf{x}$$

Using the **kernel trick** and replacing scalar products with kernel evaluations leads to

$$\widehat{f}(\mathbf{x}) = \sum_{l=1}^{n} \widehat{\alpha}^{(l)} k(\mathbf{x}_l, \mathbf{x})$$

for kernel ridge regression.

KRR

What was missing in functional gradient descent was a smoothness penalty on f. In KRR, we impose such a constraint through the kernel choice and the penalty on α.

To allow for smooth interpolations between the training data observations, we impose a constraint on the inner product of the functions f, f that are generated by this kernel choice.

We use $f(.) = \sum_i \alpha_i k(x_i, .)$ as our functional form and consider the space of all such functions:

$$G = \{\sum_i \alpha_i k(x_i, .), \alpha \in R\}$$

KRR

If we want to evaluate the function at $x : f(x)$, we can do so directly or by taking the inner product with the kernel function $k(x, .)$:
$f(x) =< f, k >=< \sum_i \alpha_i k(x_i, .), k(x, .) >= \sum_i \alpha_i < k(x_i, .), k(x, .) >= \sum_i \alpha_i k(x_i, x)$
through definition of the inner product of the kernels in the RKHS.
Now, with this representation of f we arrive at the smoothness penalty

$$||f||^2 =< \sum_i \alpha_i k(x_i, .), \sum_j \alpha_j k(x_j, .) >= \sum_{i,j} \alpha_i < k(x_i, .), k(x_j, .) >= \alpha' K \alpha$$

KRR with gradient descent

While KRR has a closed form solution, one might want to consider other losses (e.g. logistic) or penalties (e.g. sparse kPCA or KRR). Also, this gives a preview to GBM and NN type estimation.

KRR with gradient descent

If we again use the L2 loss and try estimate the function $f = K\alpha$ with penalty $\alpha' K\alpha$ with vanilla gradient descent, here are the steps;

1. Initialize with $\alpha_{i,t} = 0, \forall i, t = 0$
2. Compute the pointwise gradients w.r.t. α from the L2 $= .5||y - K\alpha||^2 + \lambda\alpha' K\alpha$ as
$$grad = -K(y - K\alpha) + \lambda K\alpha$$
3. Update $\alpha_{t+1} = \alpha_t - \eta\ grad$
4. Update $f_{t+1} = K\alpha_{t+1}$

For an out-of-training prediction, compute the rectangular kernel $K(x*, x)$ and predict $f(x*) = K(x*, x)\alpha$

Kernel logistic regression with gradient descent

Let us consider the logistic loss in functional gradient descent:

$$L = \prod_i f_i^{y_i}(1 - f_i)^{1-y_i}$$

Taking logs we obtain

$$l = \sum_i y_i log(f_i) + (1 - y_i) log(1 - f_i)$$

The derivative wrt to f of the loss-function gradients are

$$(\frac{dl}{df})_i = \frac{y_i}{f_i} - \frac{1 - y_i}{1 - f_i} = \frac{y_i(1 - f_i) - f_i + f_i y_i}{f_i(1 - f_i)} = \frac{y_i - f_i}{f_i(1 - f_i)}$$

Kernel logistic regression with gradient descent

Defining $f(z) = \sigma(z) = \frac{e^z}{1+e^z}$, the sigmoid function, and taking derivative with respect to the linear predictor $z = K\alpha$, we obtain

$$\frac{d}{dz}f(z) = \frac{e^z(1+e^z) - e^z e^z}{(1+e^z)^2} = \frac{e^z}{1+e^z}\frac{1}{1+e^z} = f(z)(1-f(z))$$

Note that the derivative of the sigmoid cancels out the numerator from the loss-function gradients!

Kernel logistic regression with gradient descent

Finally, taking the derivative of the linear predictor $K\alpha$ wrt α is K.

Thus,

$$\frac{dl}{d\alpha} = \frac{dl}{df}\frac{df}{dz}\frac{dz}{d\alpha} = K(y - f) = K(y - K\alpha)$$

via the chain rule!

If we add the ridge penalty:

$$\frac{d(l + \alpha' K\alpha)}{d\alpha} = \frac{dl}{df}\frac{df}{dz}\frac{dz}{d\alpha} = K(y - (K + \lambda I)\alpha)$$

Kernel logistic regression with gradient descent

1. Initialize with $\alpha_{i,t} = 0, \forall i, t = 0$
2. Compute the pointwise gradients w.r.t. α as above

$$grad = K(y - (K + \lambda I)\alpha)$$

3. Update $\alpha_{t+1} = \alpha_t + \eta\, grad$ (note max likelihood)
4. Update $f_{t+1} = \sigma(K\alpha_{t+1})$

For an out-of-training prediction, compute the rectangular kernel $K(x*, x)$ and predict $f(x*) = \sigma(K(x*, x)\alpha)$

Forward stepwise/stagewise regression

Forward fitting

A key to avoiding overfitting is to slow down training (learning rate), stop training before the methods start to overfit or to regularize the model output in some way - or a combination of all of the above.

In high-dimensional regression, training with gradient descent, starting from 0, is a way to regularize the model output. If you stop GD early, you are essentially controlling the norm of the coefficient estimates - just like ridge regression.

Forward selection

There are other ways of preventing overfitting in a forward fitting scheme. Forward selection is a method for including one feature at a time and then update the model to include the next best feature etc:

1. Initialize with 0 features included, empty $model_0 = \emptyset$
2. Add to the model the optimal predictor

$$j* \in J = \{1, \cdots, p\}$$

   that minimizes the loss function (mse or logistic for example):

$$model_t = c(model_t, j*)$$

3. Iterate step 2 until STOP

You can STOP based on model selection criteria, validation loss, etc.

Note, all coefficients are updated in each iteration of step 2.

Forward stagewise regression

The GD updates leads to all features being included in the model from the start, whereas the forward selection algorithm includes as many features as the number of steps.

Forward stagewise regression is a method that updates the model one parameter at a time;

1. Initialize with 0 features included, $\beta(0) = 0, t = 0$, compute the residuals $r = y - X\beta_t$.
2. Find the feature j with most correlation wrt to the residuals r: $j* = \arg\max_j corr(x_j, r))$
3. Update $\beta_{j*}(t) = \beta_{j*}(t) + \eta\ sign(corr(x_{j*}, r)$
4. Iterate steps 2 and 3 until STOP

You can STOP based on model selection criteria, validation loss, etc.

Forward fitting schemes revisited

Forward fitting schemes can be adjusted further by letting the η be optimized in each fitting step.

That is the foundation for sparse regression via LARS (a lasso variant) where we travel as far along the max-corr feature we can until another feature becomes more correlated with the residuals.

This will result in fewer features being picked across the full run of the algoritm and with early stopping results in a feature selection method that is widely used today.

More on L1/sparse regression to come!

Boosting

AdaBoost

Notice how the forward fitting mechanisms used either the residuals of the current fit, or the gradients of the current fit to update the regression coefficients.

This is the principles of boosting, but we will generalize to other functions than linear regression.

The idea is to build the model, component by component, by fitting models to the residuals that are also weighted in order to reallt boost the performance of each component.

AdaBoost

The model at iteration t is denoted $F_t(x)$ and comprises model components $f_t(x)$ optimally weighted by value α_t (cf. lars vs stagewise). $y \in \{-1, 1\}$

Algorithm

1. Initialize observation weights $w_i = 1/n$
2. Fit a classifier f_t to the data using observation weights w_i
3. Compute the weighted error rate: $err_t = \sum_i w_i 1\{y_i \neq f_t(x_i)\}$
4. Compute $\alpha_t = log \frac{1-err_r}{err_t}$ (zooming in on errors)
5. Update the weights as

   $$w_{i,t+1} = w_{i,t} exp(-\alpha_t y_i f_t(x_i))/Z$$

   , where Z is used to normalize the weights to add to 1
6. Update $F_{t+1} = F_t + \alpha_t f_t$
7. Predict $y = sign(F_{t+1})$

AdaBoost

But that doesn't look like our forward stagewise methods?!?!

Actually, it does - we are just using a particular loss function here.

- Define the loss as $L(y, f) = exp(-y * f)$ and note that with $y \in \{-1, 1\}$ and the sign-prediction, $y * f$ can only take on values -1 or 1
- $L(y, F_{t-1} + \alpha f_t) = \sum_i exp(-y_i(F_{t-1} + \alpha f_t(x_i))) = \sum_i exp(-y_i F_{t-1}(x_i)) exp(-y_i \alpha f_t(x_i))$.
- Can we show that $w_{i,t} = exp(-y_i F_{t-1}(x_i))$?

AdaBoost

- Can we show that $w_{i,t} = exp(-y_i F_{t-1}(x_i))$?
- The update scheme stated that (propotionally)

$$w_{i,t} = w_{i,t-1} exp(-\alpha_t y_i f_{t-1}(x_i))$$

- Plug in the expression for $w_{i,t-1}$ as a function of $w_{i,t-2}$ and show recursively that

$$w_{i,t} = w_{i,0} exp(-y_i \sum_{t'=1}^{t-1} \alpha_{t'} f_{t'}(x_i))$$

  and with the constant initialization this proves that

$$w_{i,t} \propto exp(-y_i F_{t-1}(x_i))$$

AdaBoost

We're now going to use the fact that $y * f$ is either -1 or 1.

- Using this fact we can write the loss as

$$L_t = exp(-\alpha_t) \sum_{y_i = f_t(x_i)} w_{i,t} + exp(\alpha_t) \sum_{y_i \neq f_t(x_i)} w_{i,t} =$$

$$= exp(-\alpha_t)A + exp(\alpha_t)B$$

- Take derivatives wrt α and set to 0

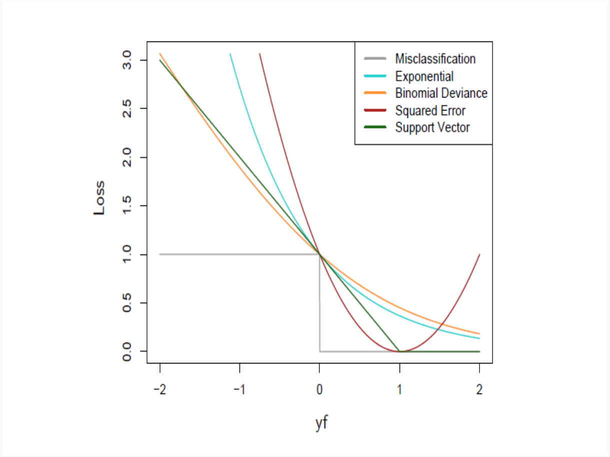
$$-exp(-\alpha)A + exp(\alpha)B = 0$$

- Take logs and re-arrange:

$$2\alpha = log(A/B)$$

$$\alpha = .5log(A/B) = .5log(\frac{1 - err_t}{err_t})$$

Loss functions



Loss functions

General idea of boosting:

1. Initialize with $F_0 = 0$
2. For each t, minimize the loss

$$L(y, F_{t-1}(x) + \alpha f_t(x, \theta))$$

   wrt α, θ

3. Update $F_t = F_{t-1} + \alpha^* f_t(x, \theta^*)$

This is a more general framework because we can actually choose the model class, parameterized by θ, in each iteration.

GBM

This is a more general framework because we can actually choose the model class, parameterized by θ, in each iteration.
In forward stagewise regression, we optimze $f(., \theta)$ through the max correlation with the residuals and features.
In *Gradient Boosting* we use so-called $pseudo-residuals$

$$r_i = -\frac{dL(y_i, F(x_i))}{dF(x_i)}|_t$$

Now this is starting to look very familiar again - we are back to functional gradient descent!

GBM

In gradient boosting (machines), we think of the negative loss gradient, r_i, as the substitute for the weighted data in AdaBoost.

Here, the magnitude of the loss gradient tells us how far off we are with the model at this iteration and we should focus on the largest gradients.

To make this gradient update work for prediction, we parameterize the model approximation of the gradients through weak learners like shallow trees!

GBM

Algorithm:

1. Initialize with $F_0 = 0$ everywhere
2. For each iteration t, compute

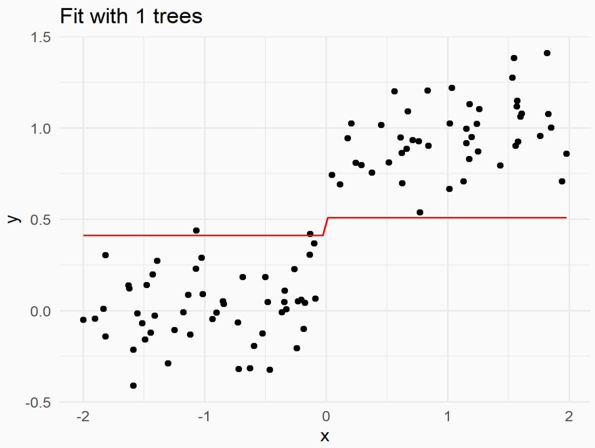
$$r_i = -\frac{dL(y_i, F(x_i))}{dF(x_i)}|_t$$

3. Fit a weak learner (e.g. shallow tree) $f_t(x, \theta)$ to the residuals
4. Optimize the model update:

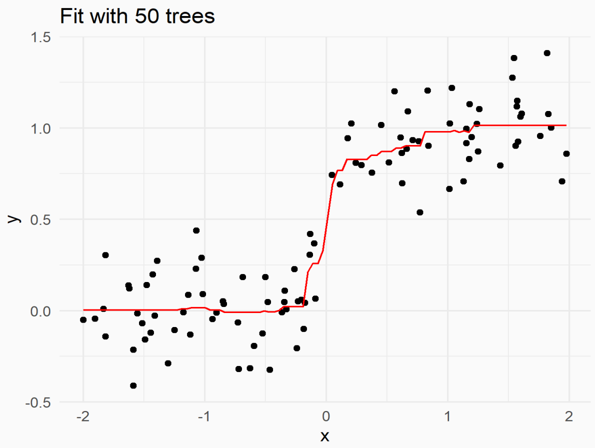
$$\arg\min_{\alpha} L(y, F_{t-1} + \alpha f_t(x, \theta^*))$$

5. Update $F_t(x) = F_{t-1}(x) + \alpha^* f_t(x, \theta^*)$

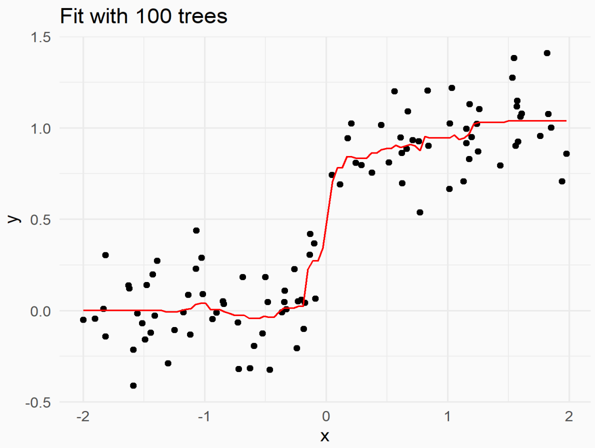
GBM



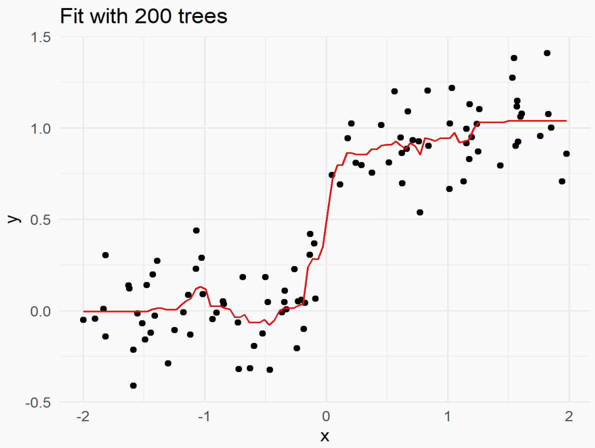
GBM



GBM



GBM



GBM

And now we’re good to go!

You can choose different weak learners and loss functions!!!

When do we stop adding model components?