Intro to Gradient Tree Boosting

Introduction

Goal

- Ensemble model
- Component models are diverse

Previous Strategies

1. Pick models that are different from each other in some way:
    - different model structure
    - different training sets (bagging)
    - different use of features
2. Estimate the models totally separately from each other
3. Put them together by averaging, majority vote, or stacking

Specific example: random forests

- Each tree used a different training set (bagged features)
- Each tree uses a random subset of features in searching for each split.
- The trees are all estimated separately, then predictions are averaged later.
- For example, for regression:

$$\hat{f}^{(ensemble)}(x_i) = \frac{1}{B}\sum_b \hat{f}^{(b)}(x_i)$$

Here, $\hat{f}^{(b)}(x_i)$ represents the prediction from one tree in the forest;

$\hat{f}^{(ensemble)}(x_i)$ is the random forest prediction.

New Strategy: Boosting

Boosting takes a sequential approach to estimation:

1. Start with a simple initial model (e.g., for regression start by predicting the mean).
2. Repeat the following:
    a. Fit a model that is specifically tuned to training set observations that the current ensemble does not predict well
    b. Update the ensemble by adding in this new model

Why is this a good idea?

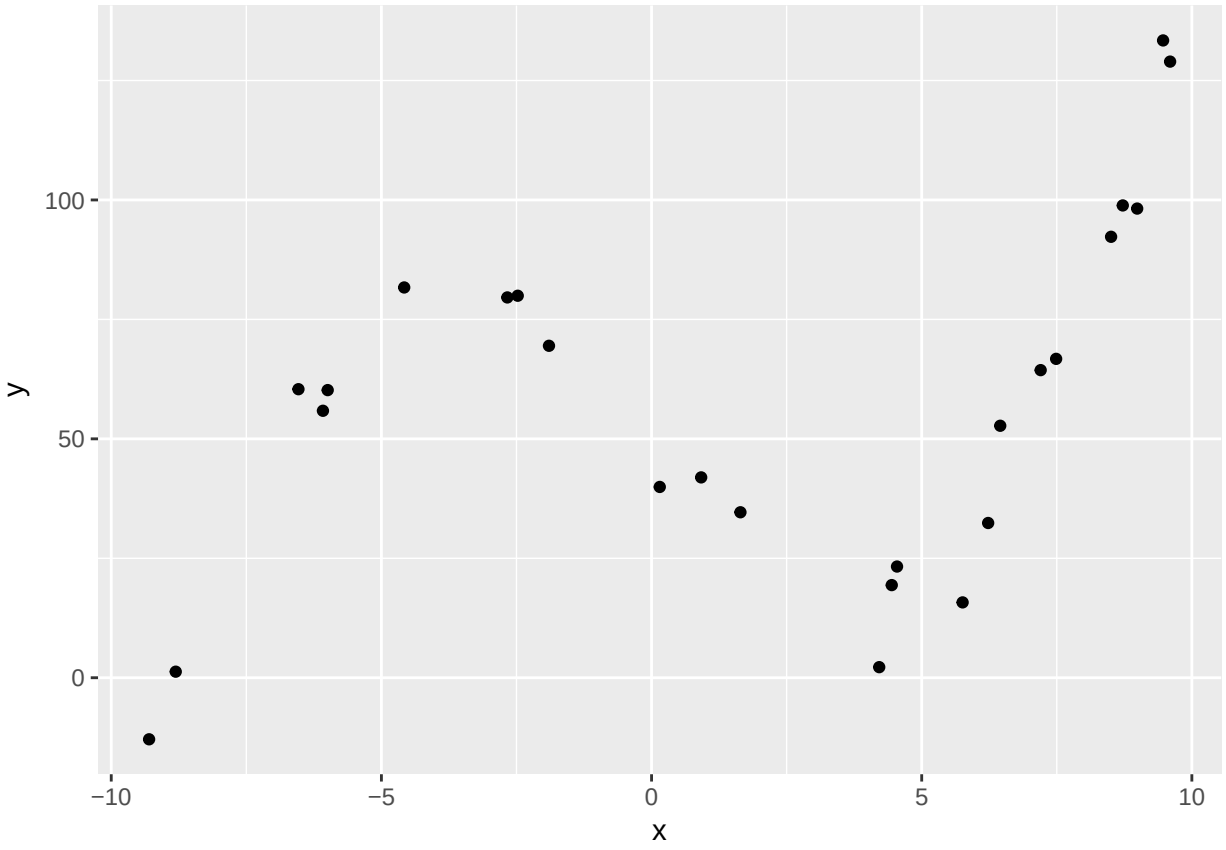
- Component models are specifically different from what's already in the ensemble!

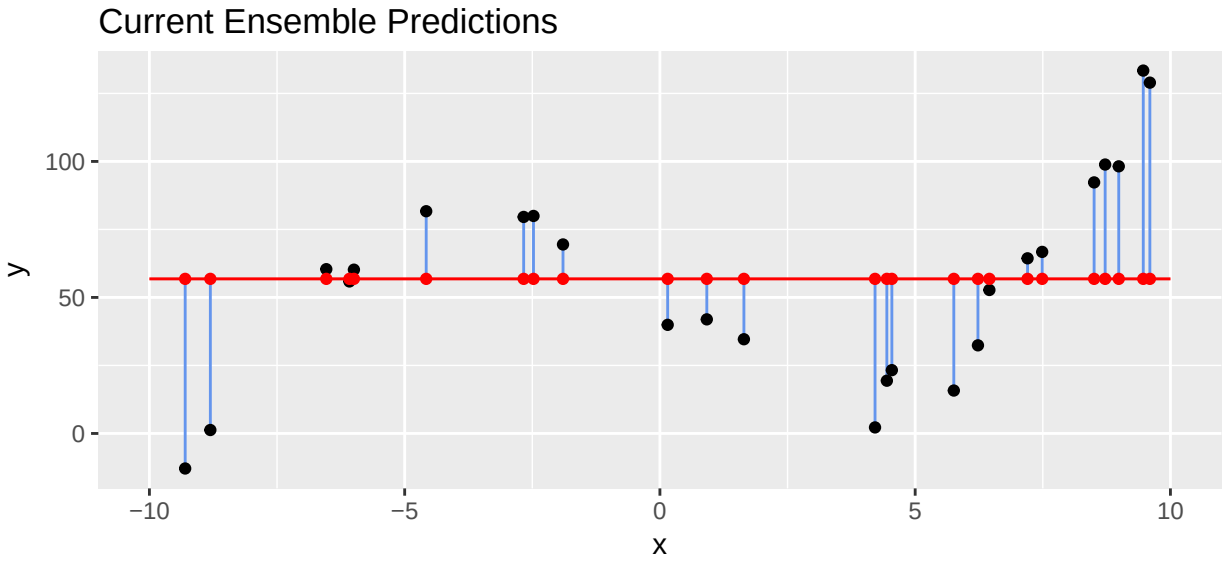
A Specific Example: Gradient Tree Boosting

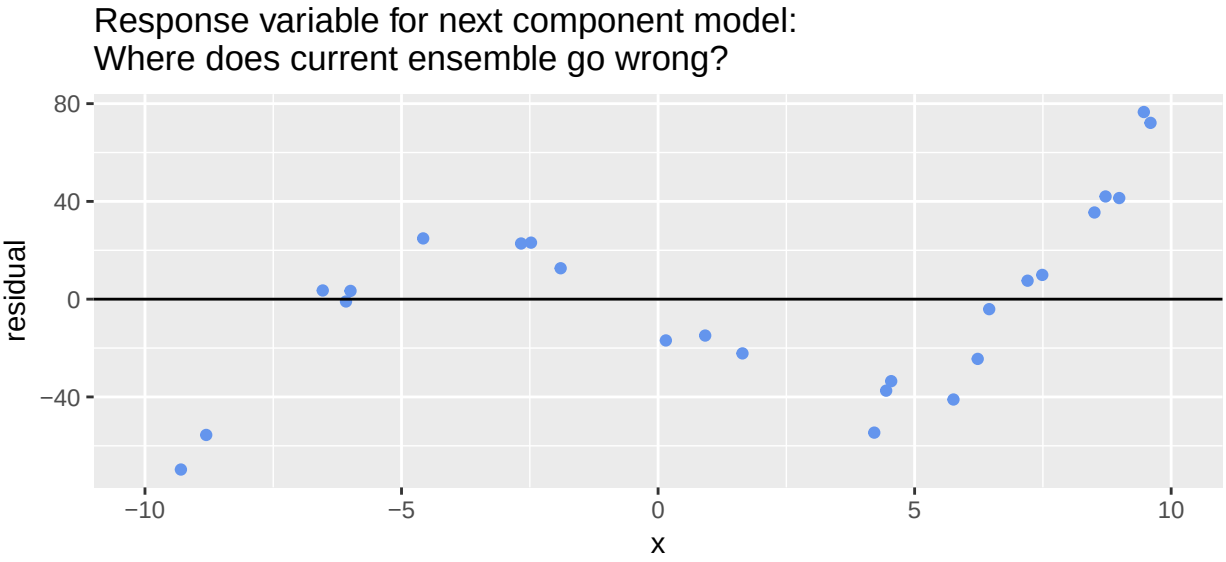
Let's start with building some intuition for the method, and define it more carefully later.

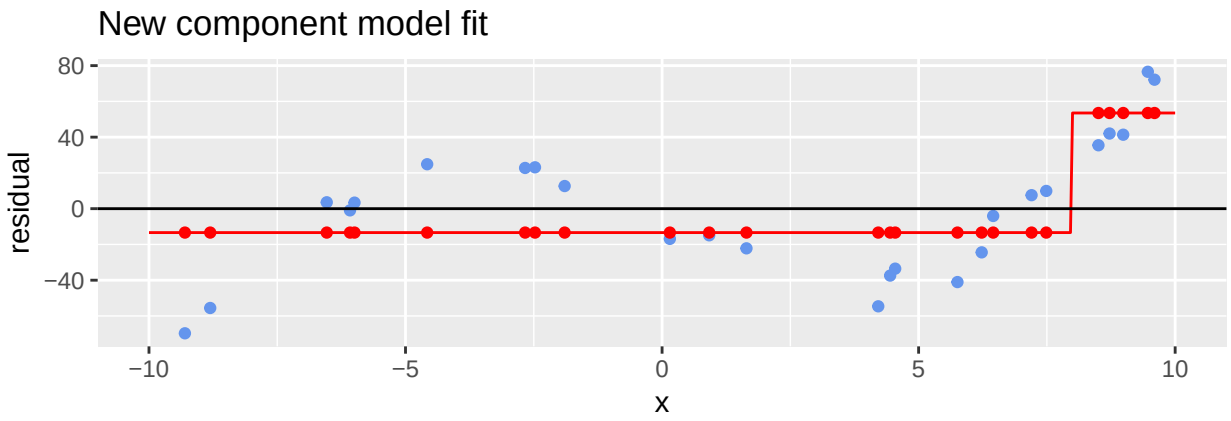
In this example, our component models will be "stumps": trees with only one split.

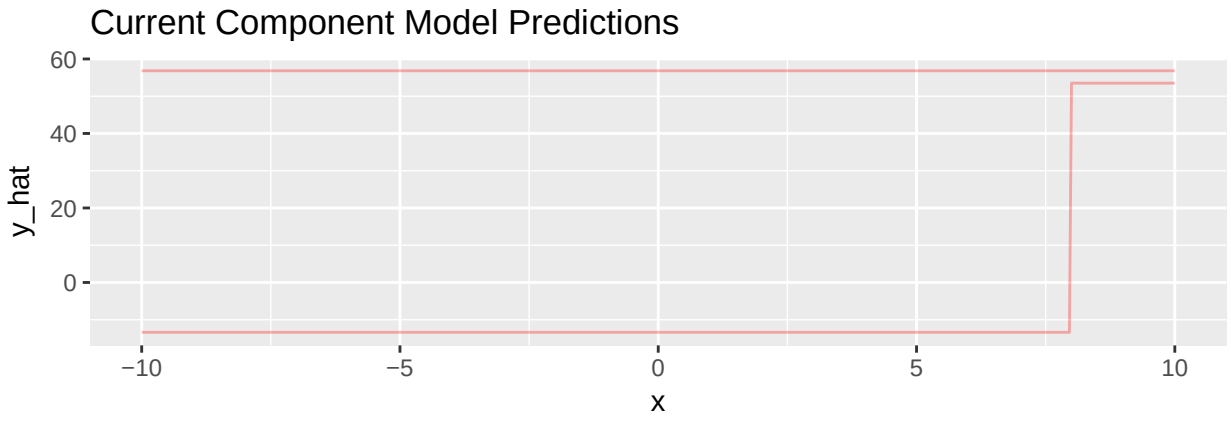
Here's a made up regression problem, and an initial prediction for each observation, given by the sample mean for the response variable.

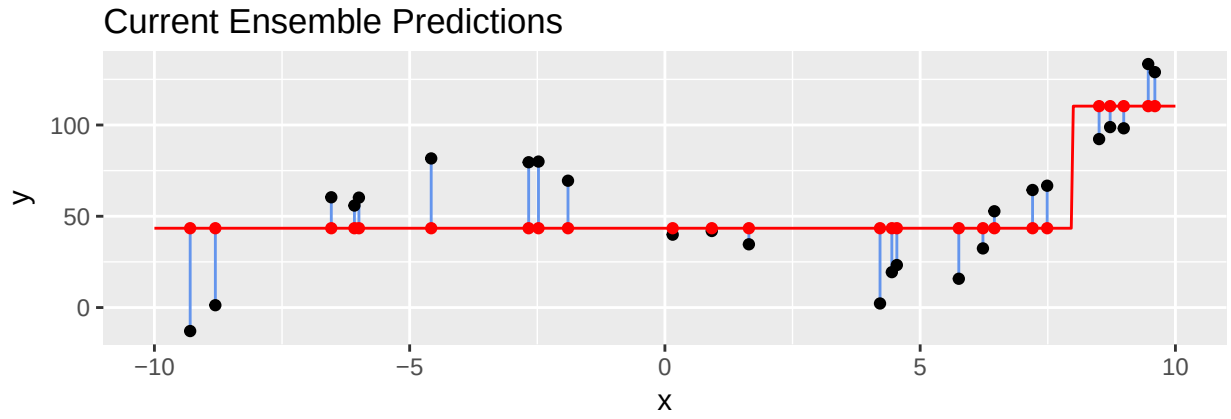


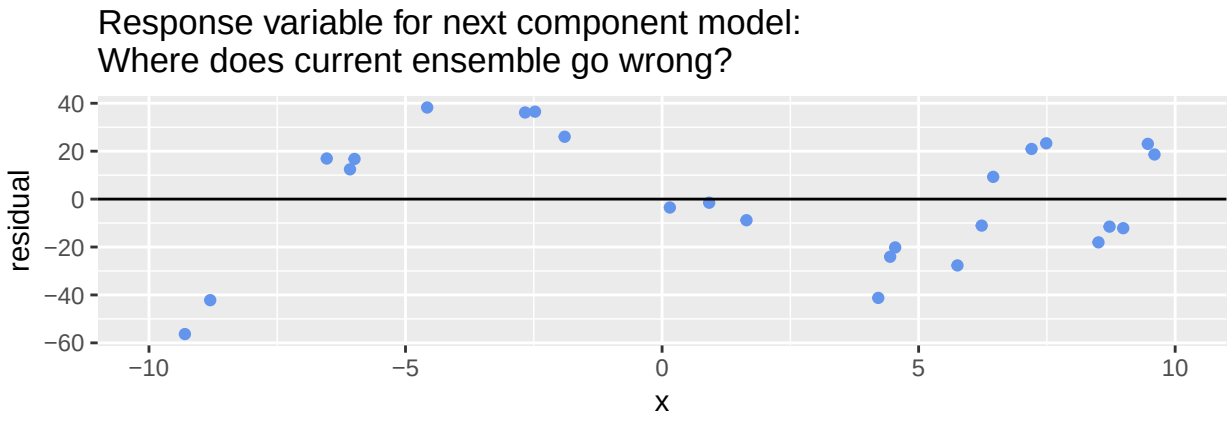

Current Ensemble Predictions
100
50
0
y
−10
−5
0
5
10
x

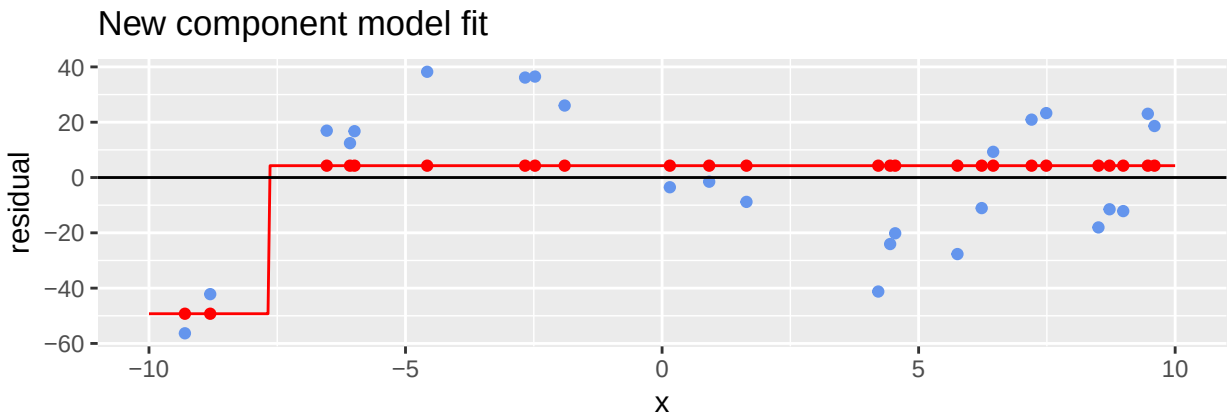

Response variable for next component model:
Where does current ensemble go wrong?
80
40
0
−40
residual
−10
−5
0
5
10
x

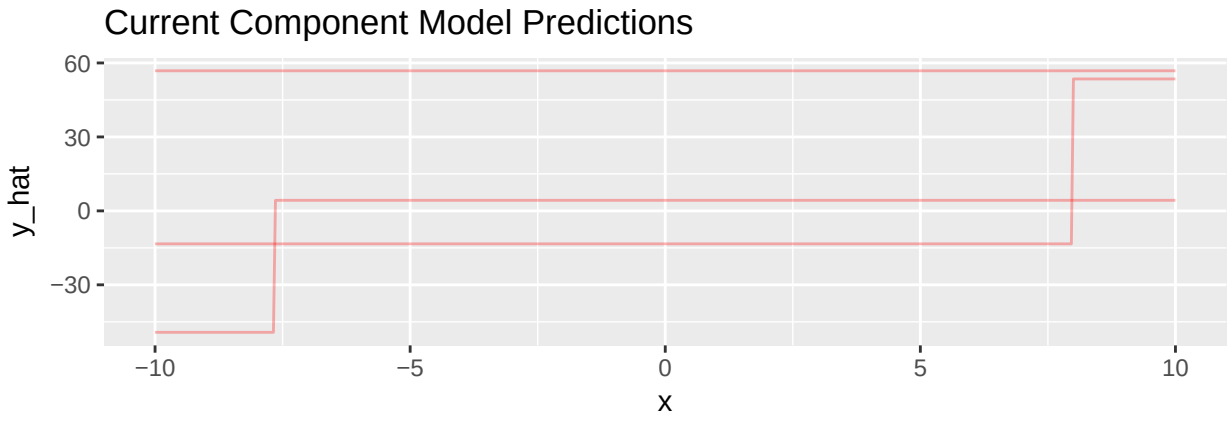

New component model fit
residual
80
40
0
−40
−10
−5
0
5
10
x

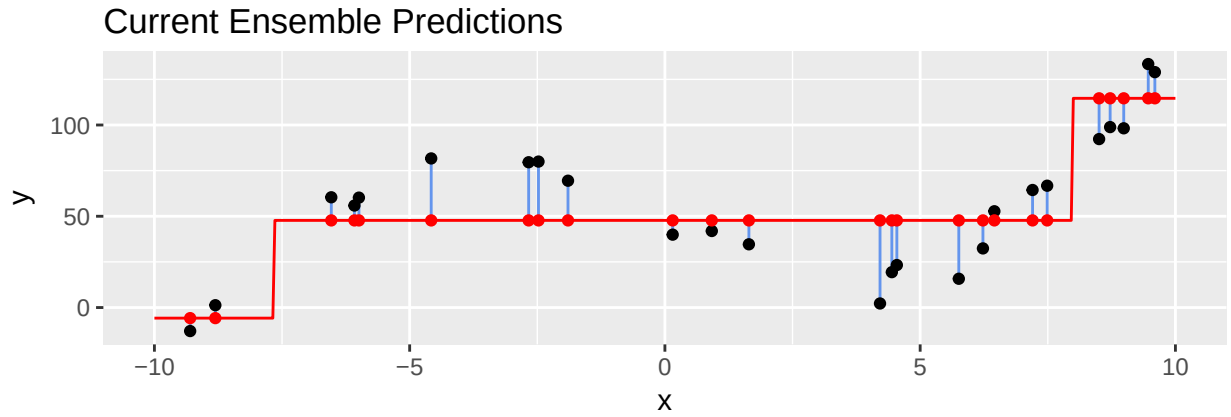

Current Component Model Predictions
y_hat
60
40
20
0
−10
−5
0
5
10
x

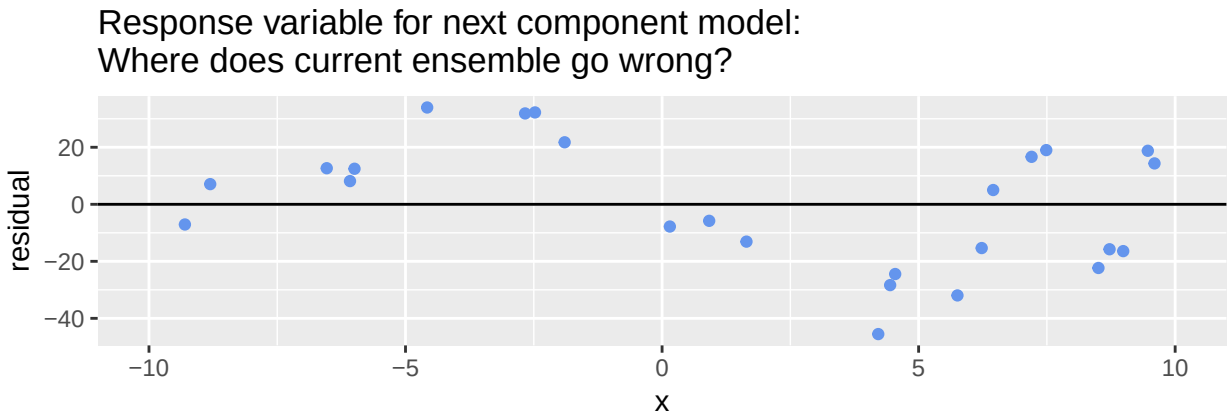

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

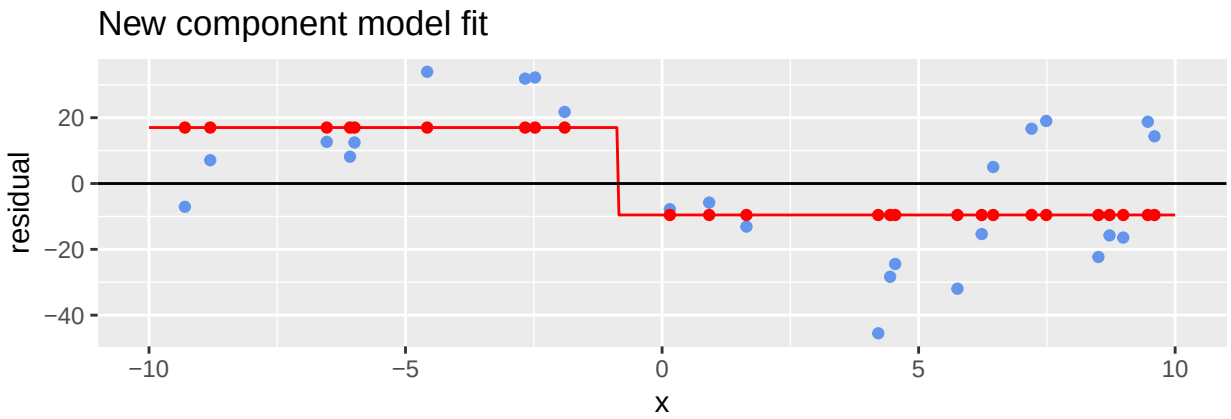

Response variable for next component model:
Where does current ensemble go wrong?
residual
40
20
0
−20
−40
−60
−10
−5
0
5
10
x

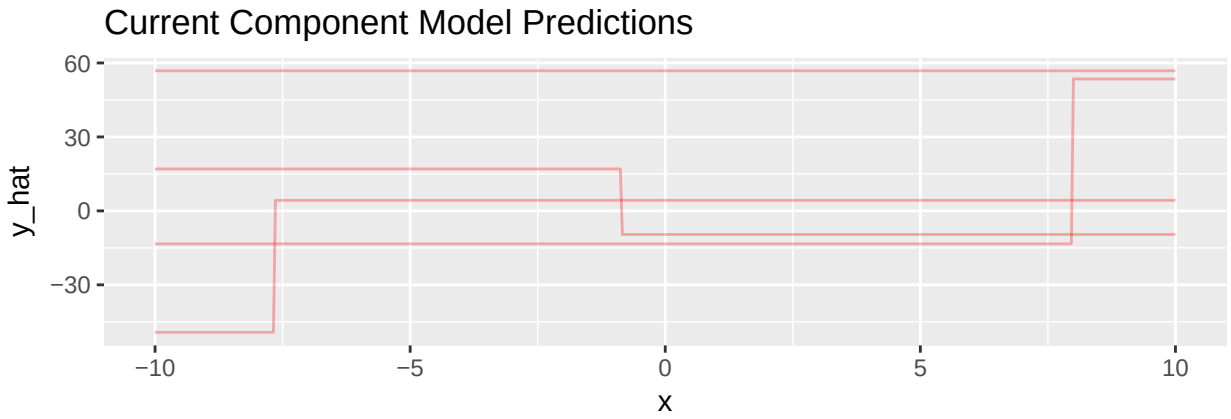

New component model fit
residual
x
40
20
0
−20
−40
−60
−10
−5
0
5
10

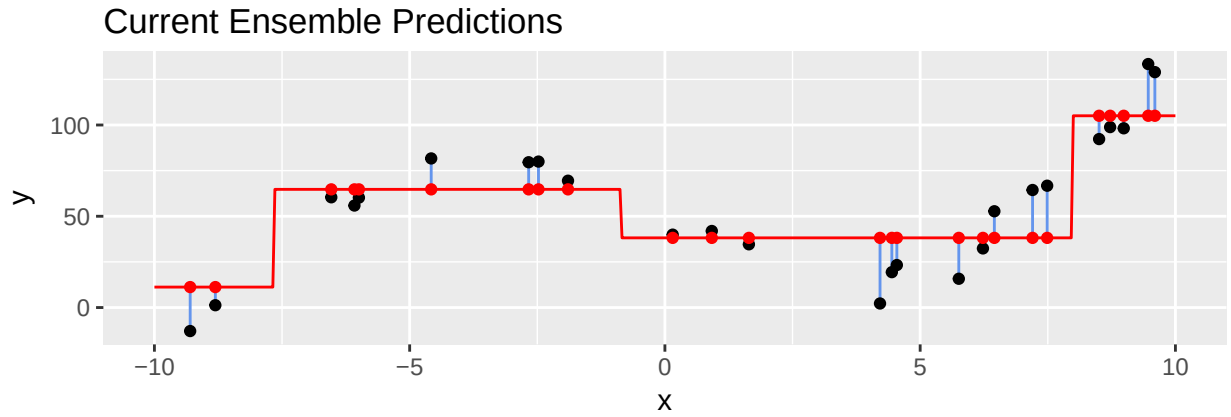

Current Component Model Predictions
y_hat
x
60
30
0
−30
−10
−5
0
5
10

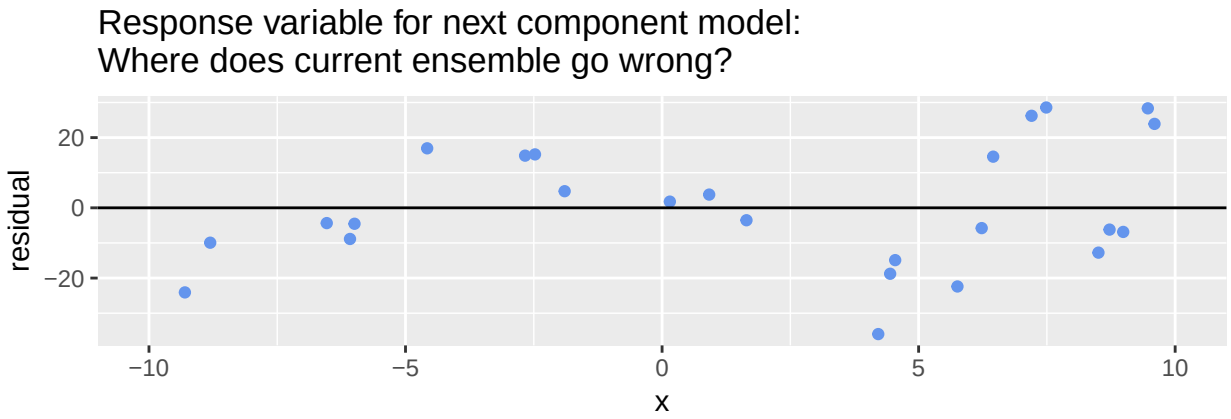

Current Ensemble Predictions
y
x
100
50
0
−10
−5
0
5
10

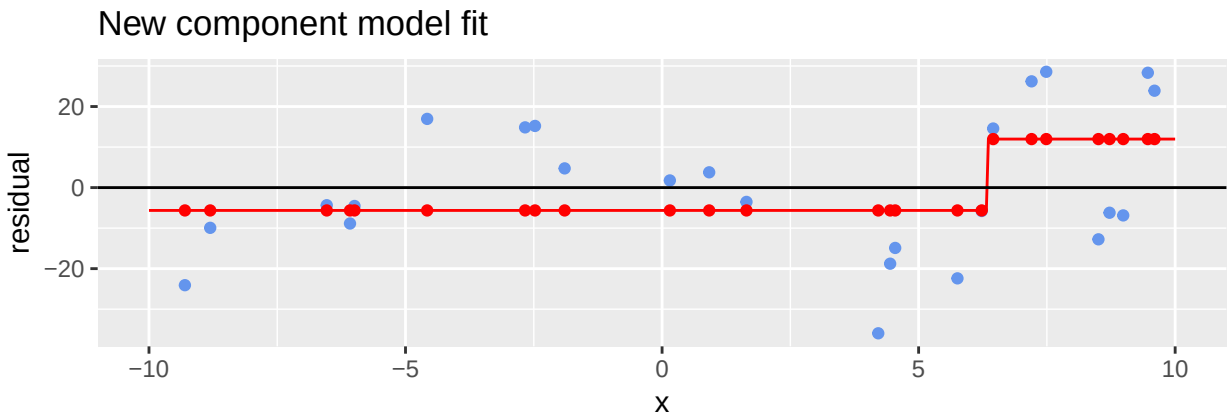

Response variable for next component model:
Where does current ensemble go wrong?
residual
x
20
0
−20
−40
−10
−5
0
5
10

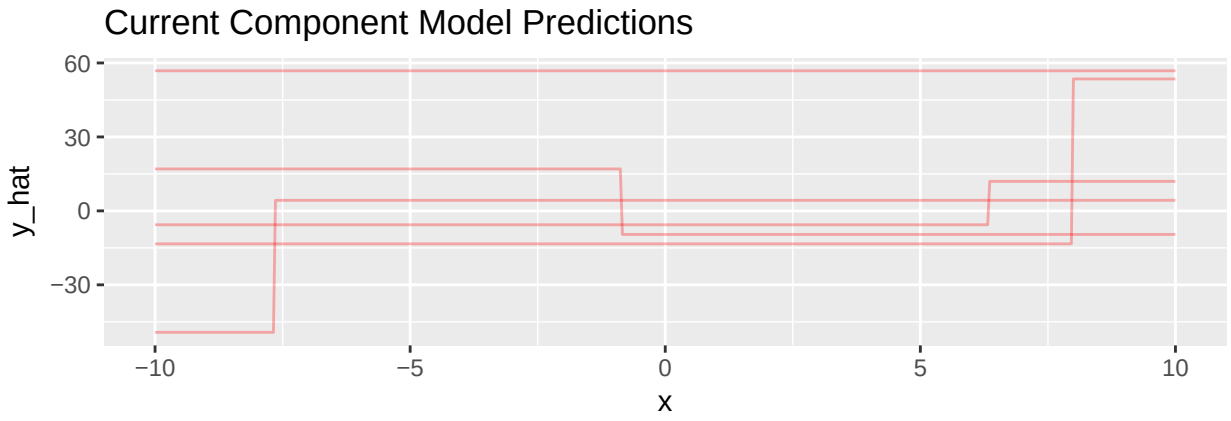

New component model fit
residual
20
0
−20
−40
−10
−5
0
5
10
x

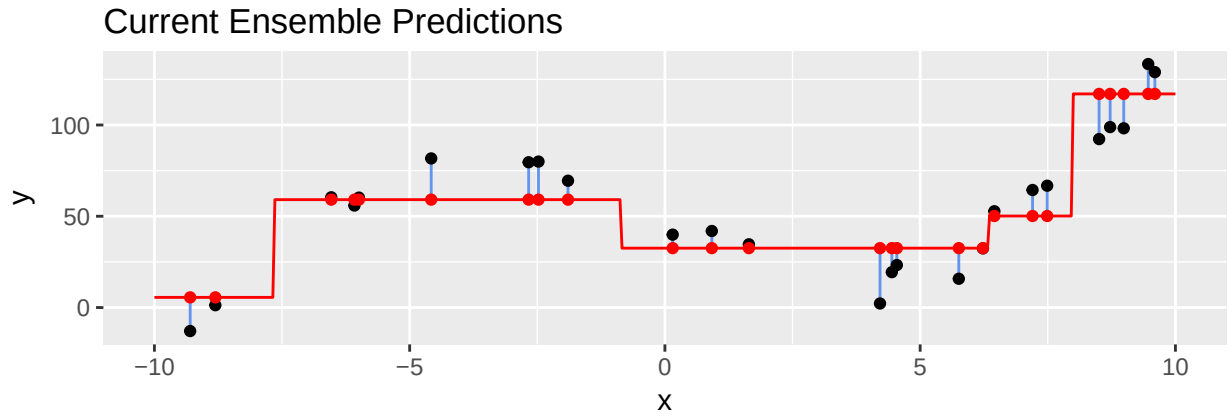

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

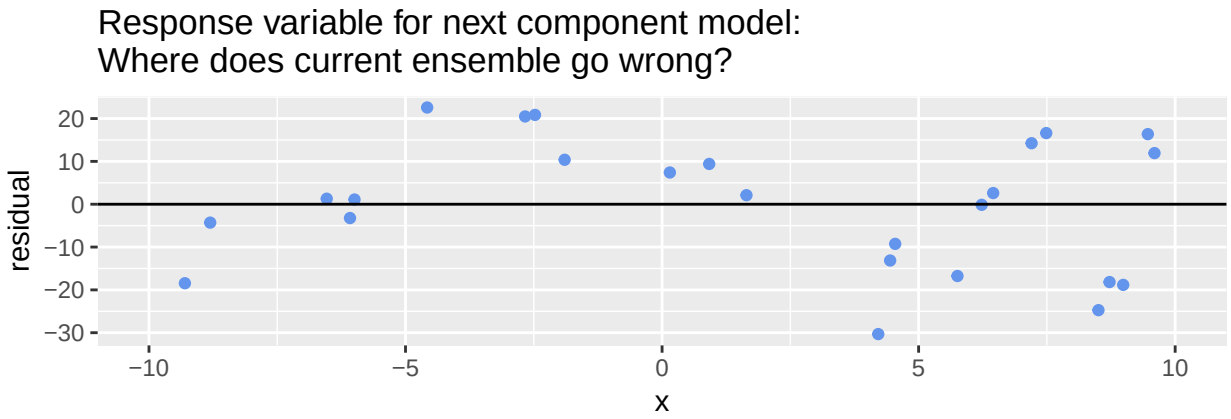

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x


Response variable for next component model:
Where does current ensemble go wrong?
residual
20
0
−20
−10
−5
0
5
10
x

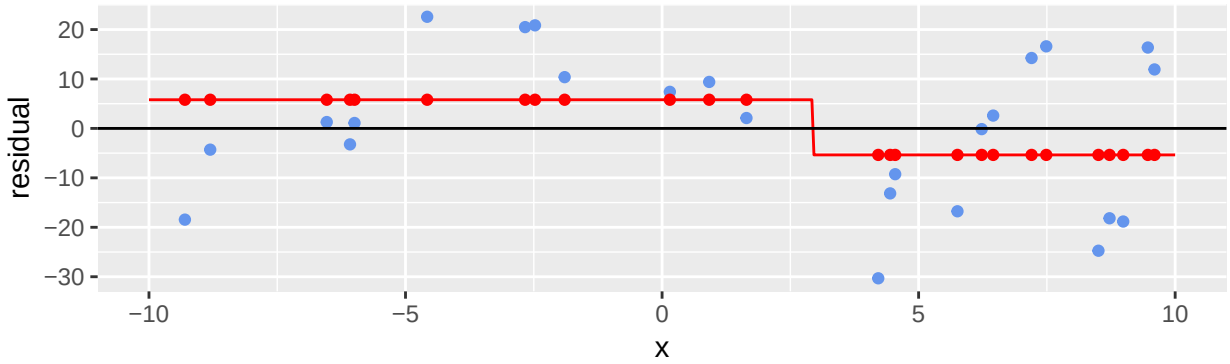

New component model fit
residual
20
0
−20
−10
−5
0
5
10
x


Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

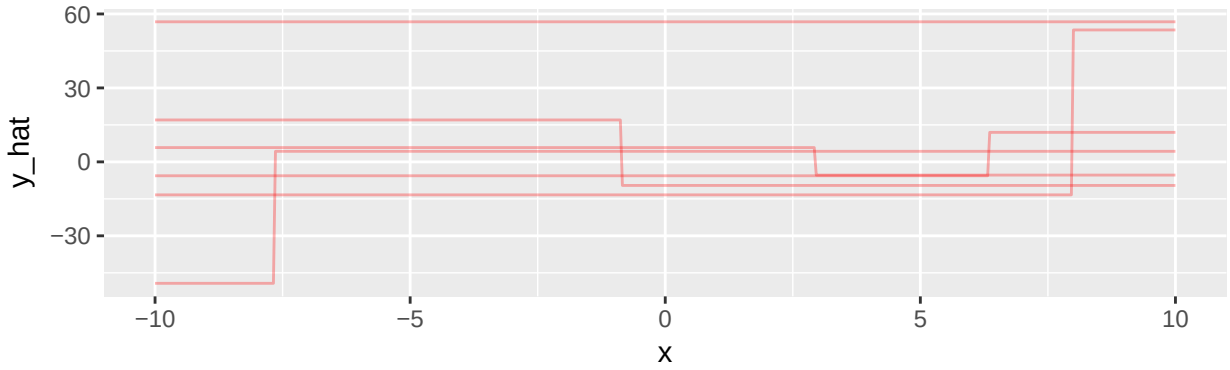

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x


Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−30
−10
−5
0
5
10
x

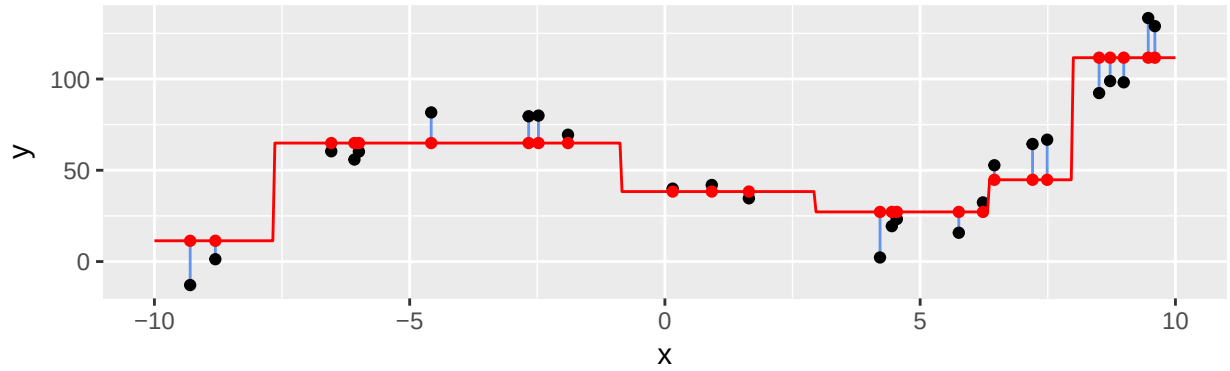
New component model fit



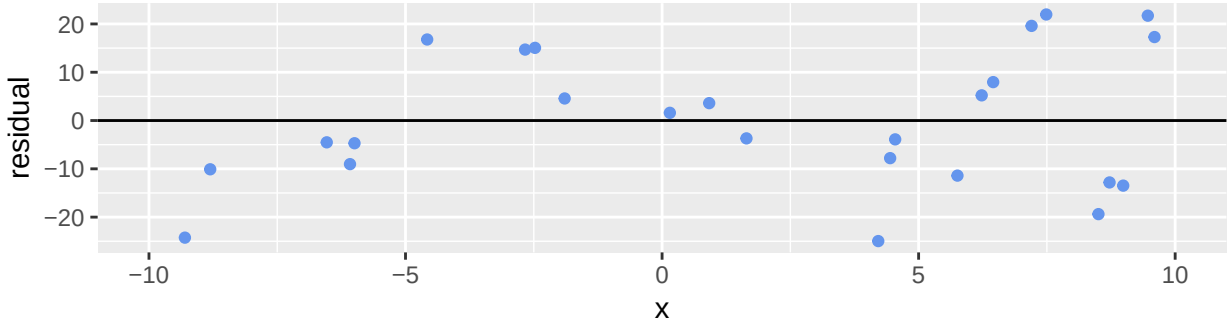
Current Component Model Predictions



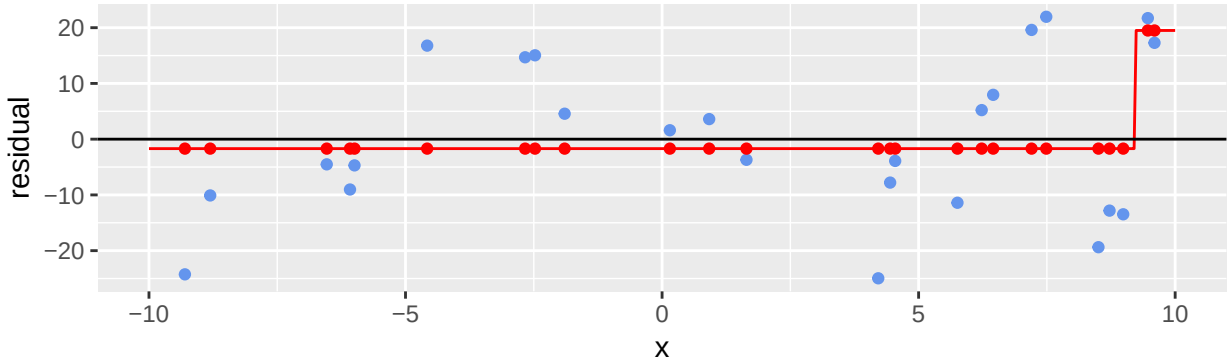
Current Ensemble Predictions



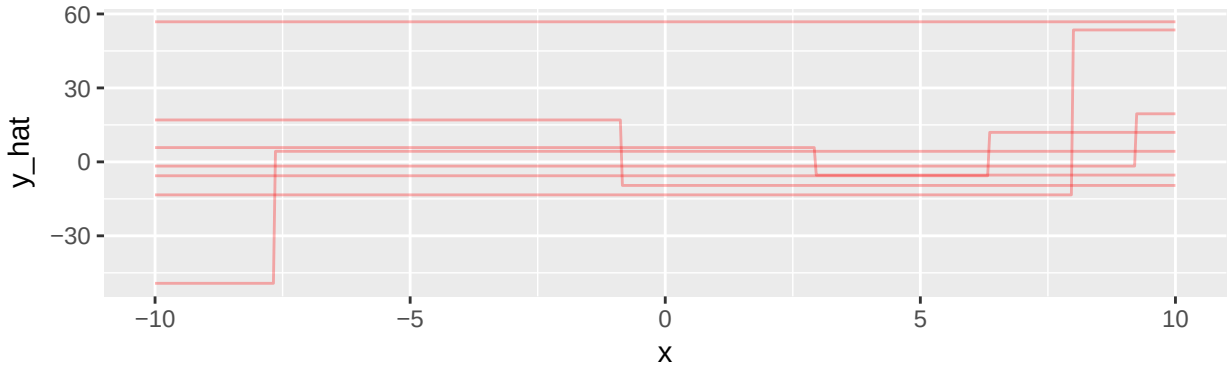
Response variable for next component model:
Where does current ensemble go wrong?



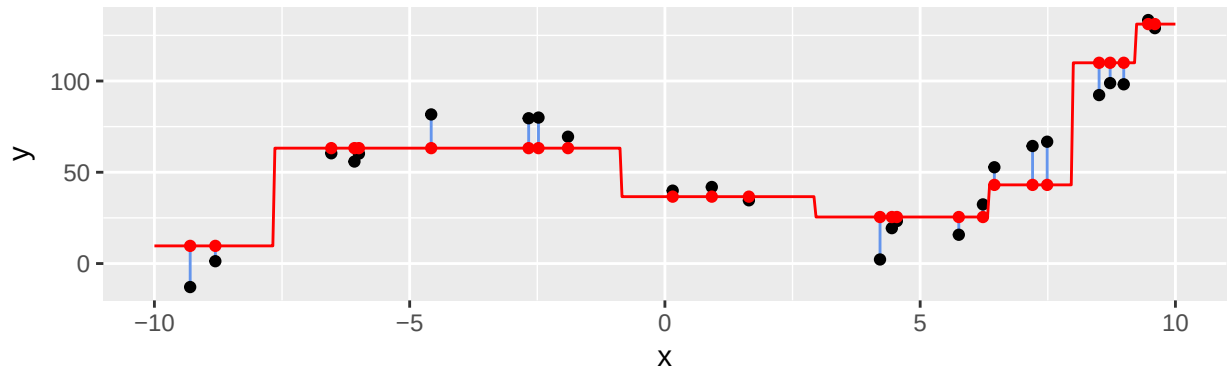
New component model fit



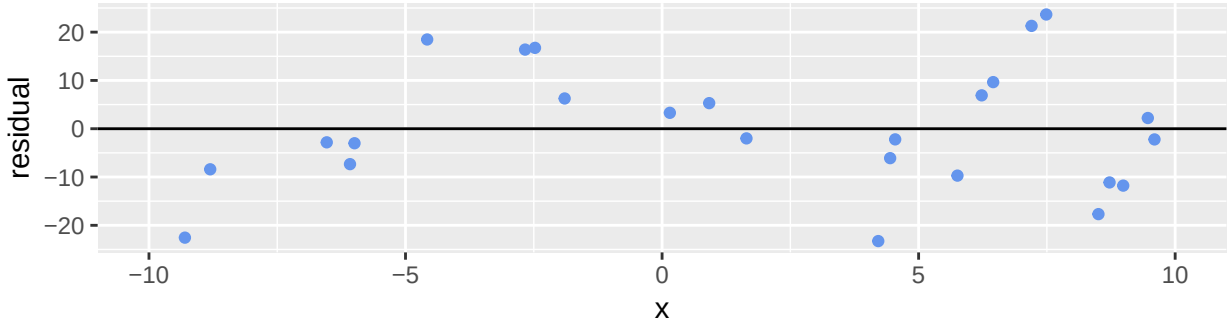
Current Component Model Predictions

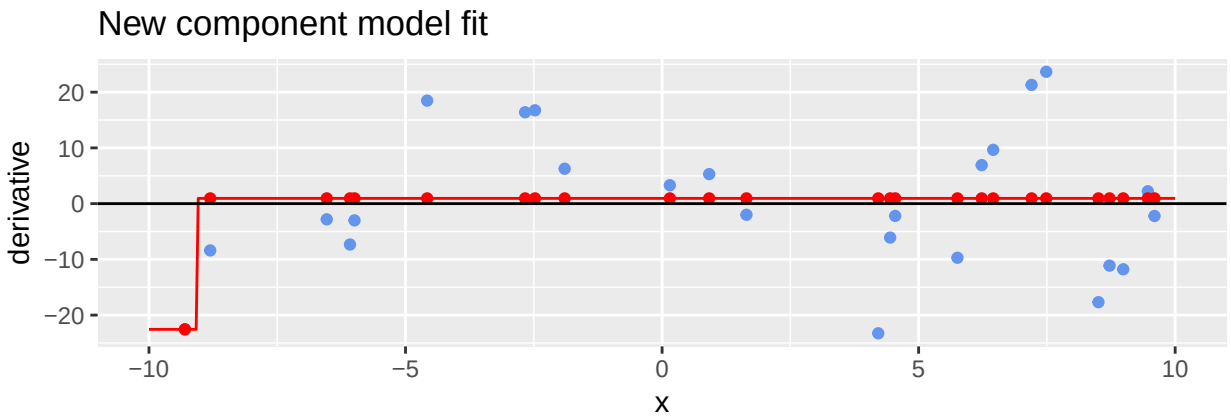


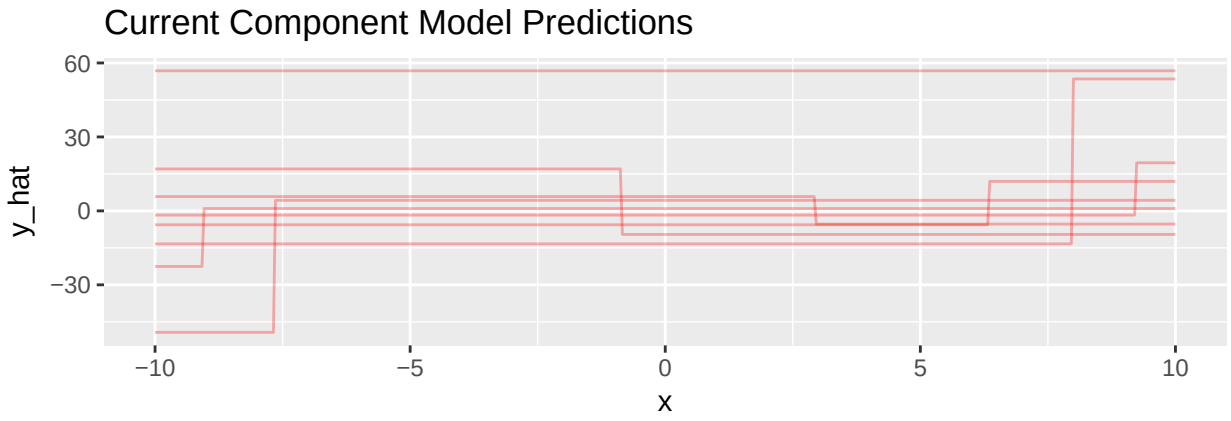
Current Ensemble Predictions

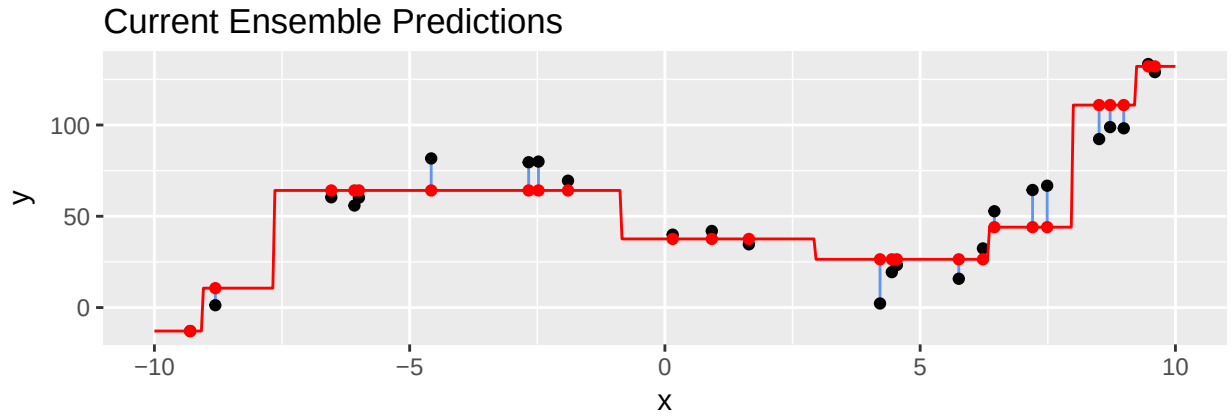


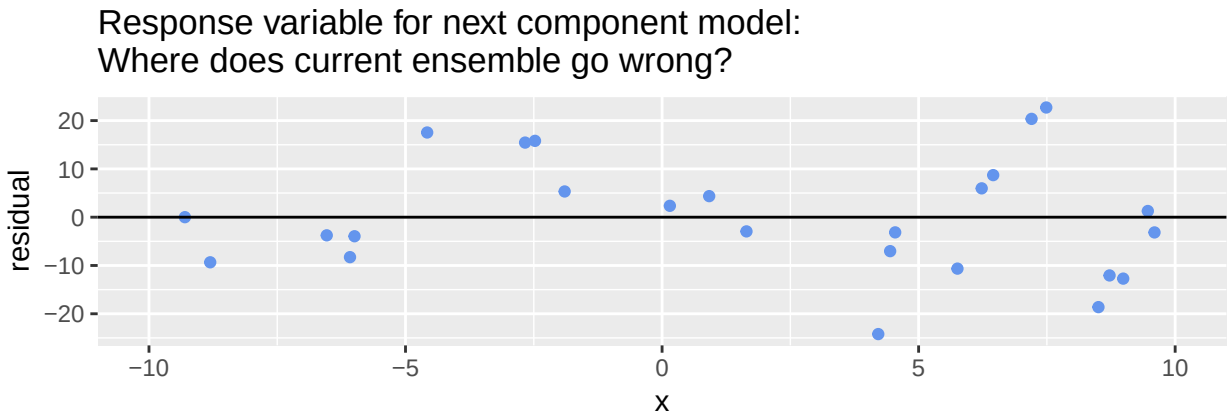
Response variable for next component model:
Where does current ensemble go wrong?

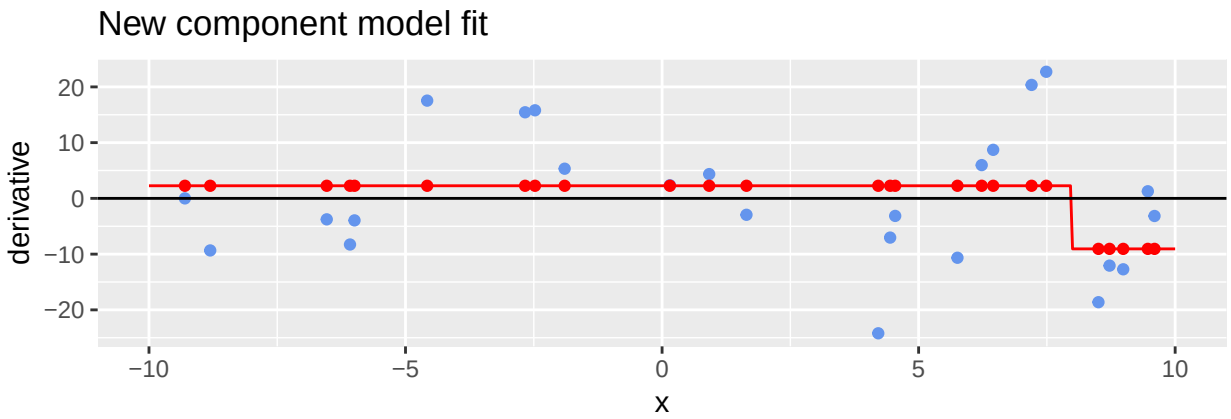


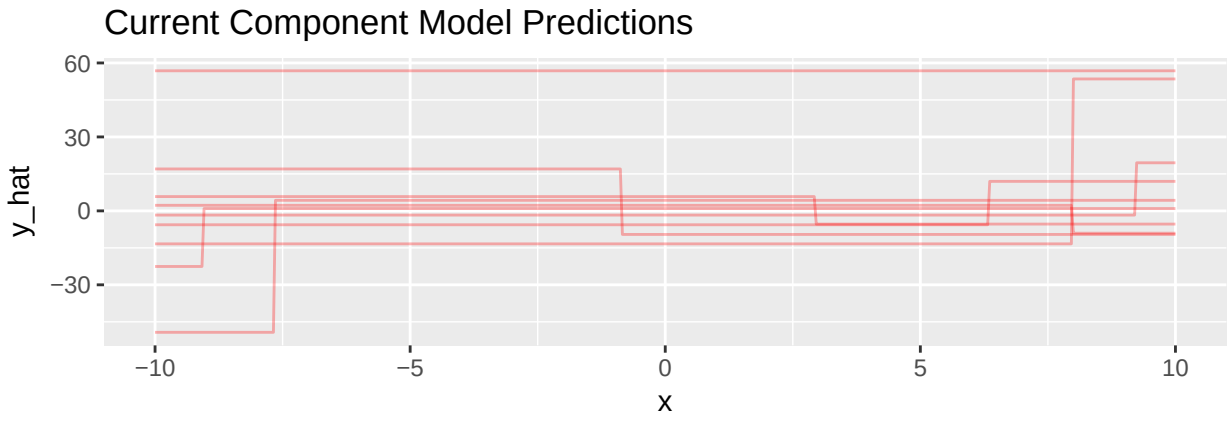

New component model fit
derivative
20
10
0
−10
−20
−10
−5
0
5
10
x

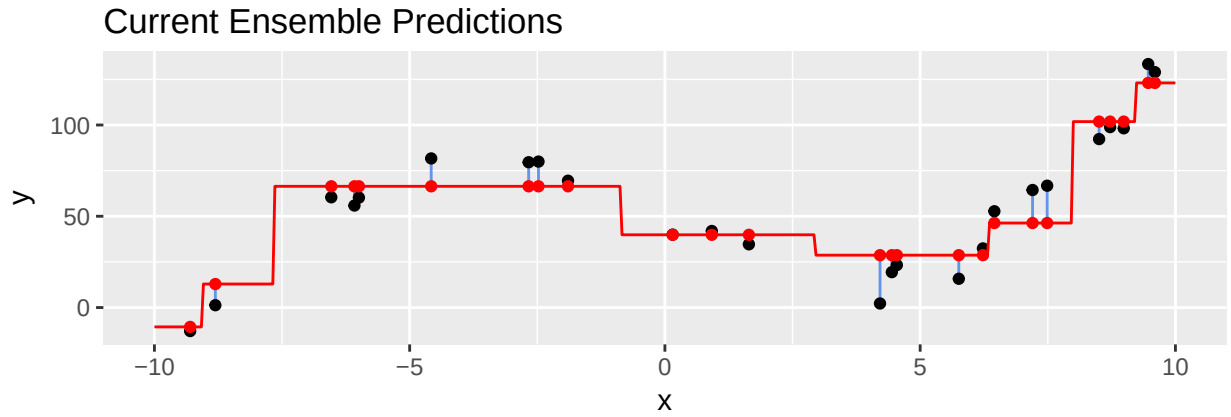

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

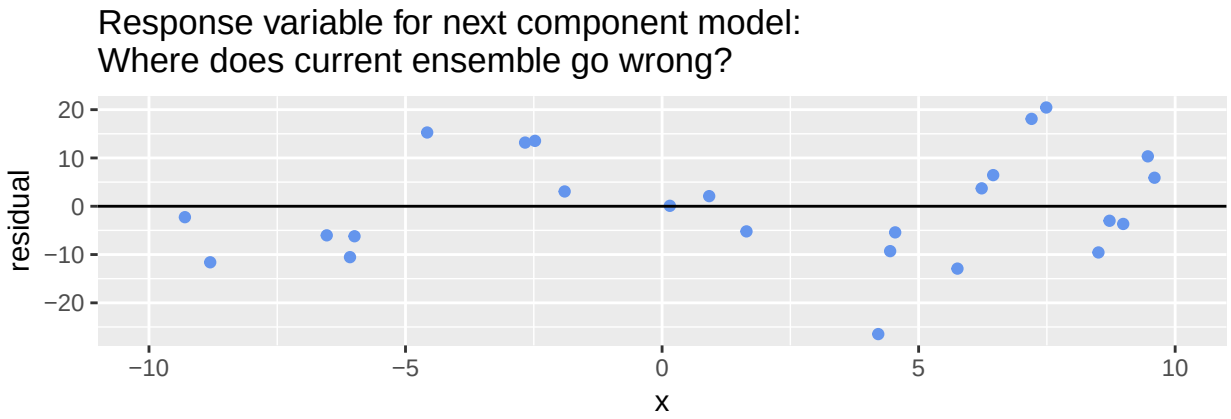

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

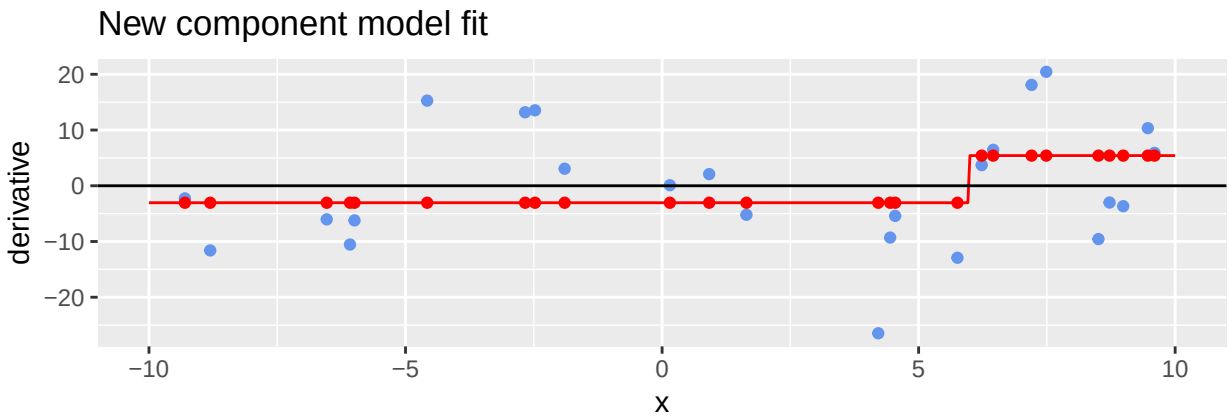

Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−10
−5
0
5
10
x

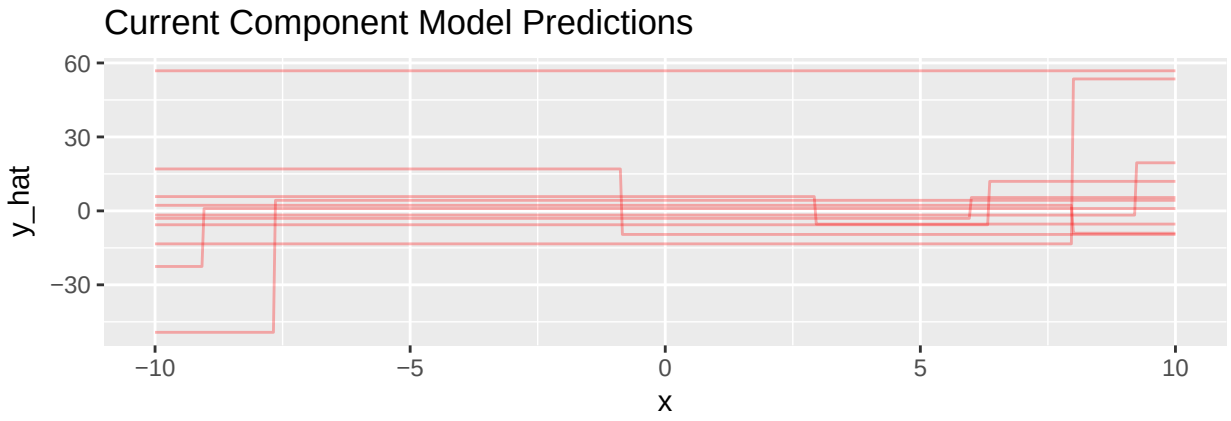

New component model fit
derivative
20
10
0
−10
−20
−10
−5
0
5
10
x

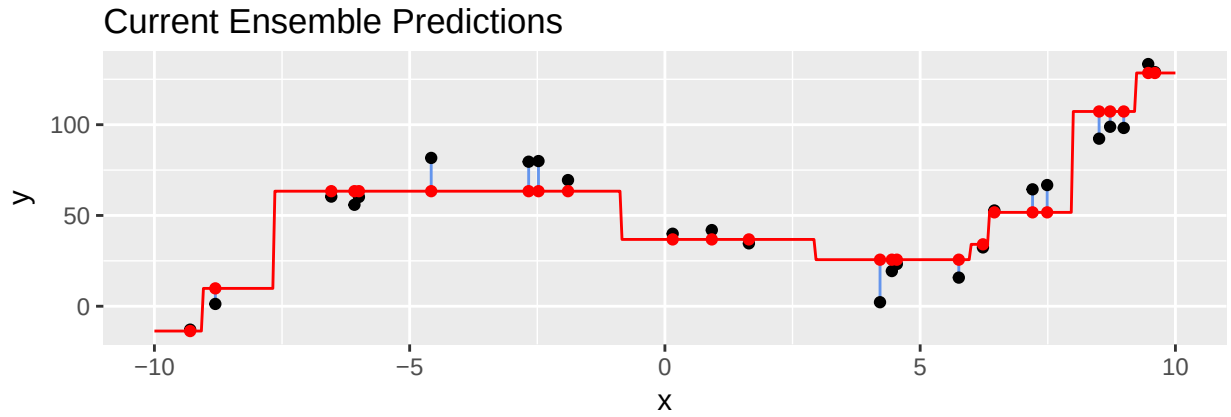

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

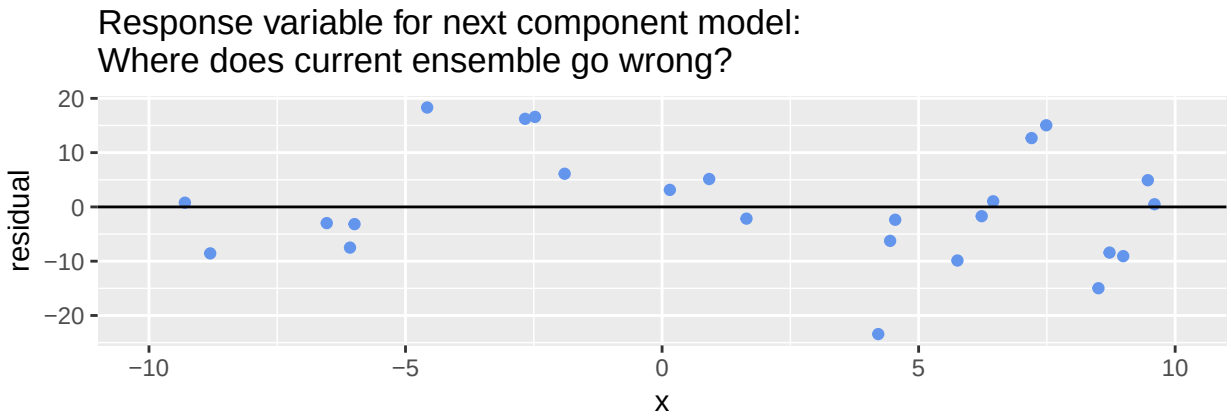

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

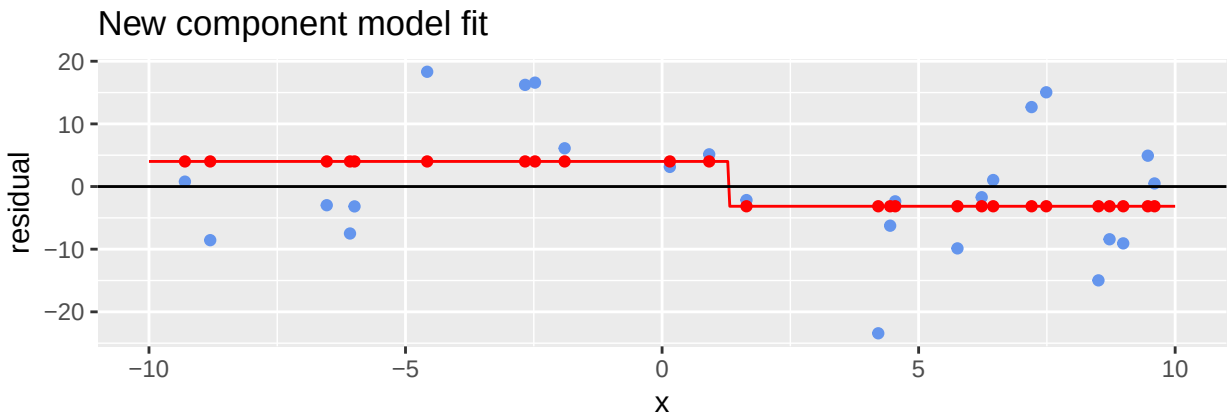

Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−10
−5
0
5
10
x

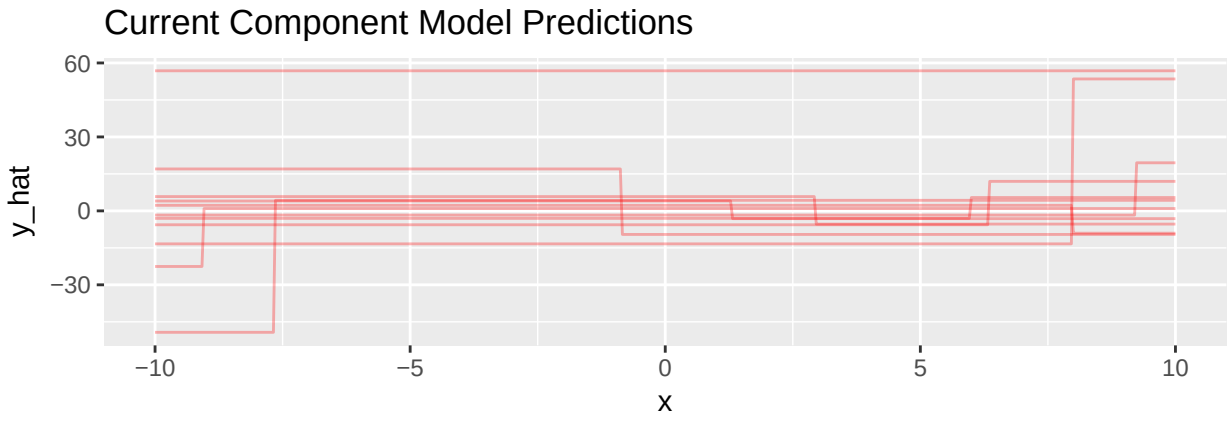

New component model fit
derivative
20
10
0
−10
−20
−10
−5
0
5
10
x

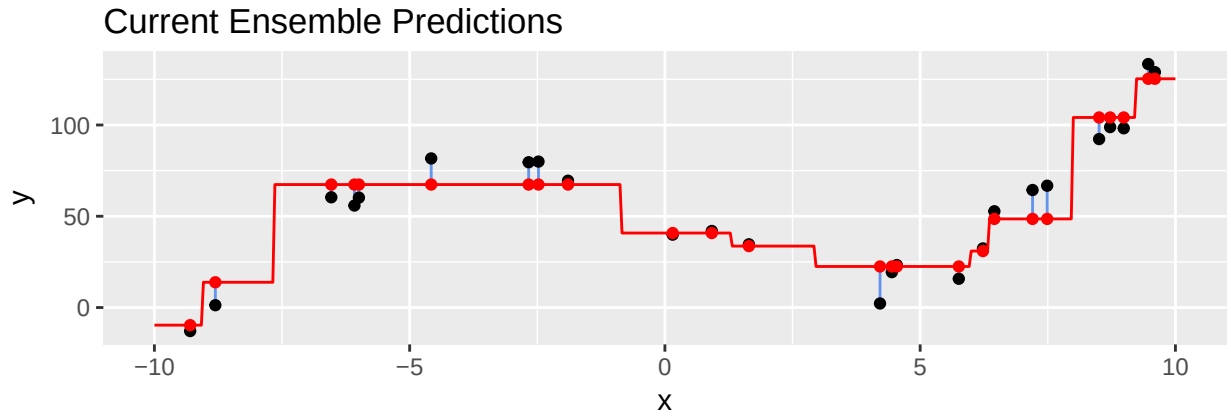

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

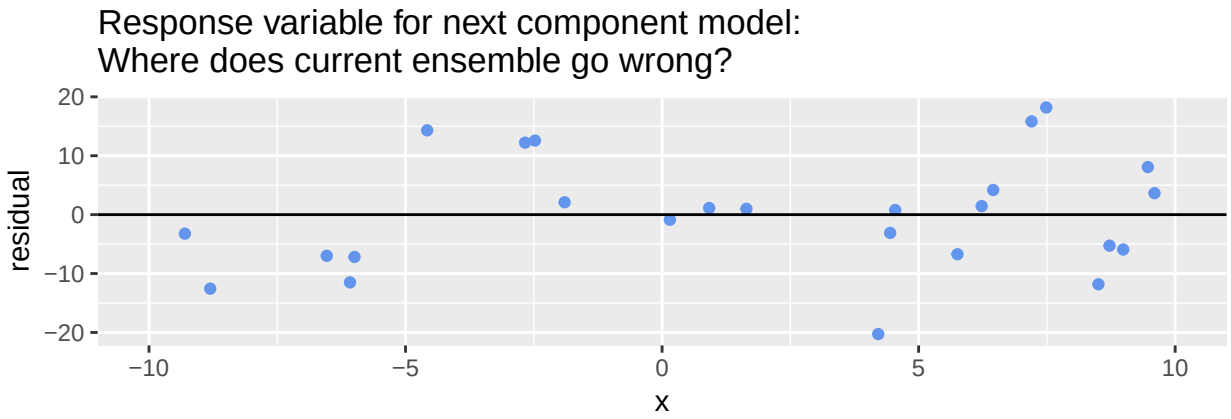

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x


Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−10
−5
0
5
10
x


New component model fit
residual
x
20
10
0
−10
−20
−10
−5
0
5
10


Current Component Model Predictions
y_hat
x
60
30
0
−30
−10
−5
0
5
10


Current Ensemble Predictions
y
x
100
50
0
−10
−5
0
5
10


Response variable for next component model:
Where does current ensemble go wrong?
residual
x
20
10
0
−10
−20
−10
−5
0
5
10

Mathematical Details: Gradient Boosting for Regression

In step b of the gradient boosting process, we have a current "working" ensemble that gives predictions $\hat{y}_i = \hat{f}^{(ensemble,b)}(x_i)$.

Above, we motivated our procedure by fitting the next component model to the residuals from the current working ensemble. Let's see how we could arrive at that procedure from a more general approach (that will also be useful for classification problems).

Set up in terms of optimizing RSS

- We want to *minimize* the residual sum of squares

$$RSS = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- ... but for our purpose now it's easier to think about *maximizing* the negative of the residual sum of squares:

$$-RSS = -\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  - If a particular function $\hat{f}(x_i)$ gives predictions that minimize RSS on the training set, then
  - $\hat{f}(x_i)$ also maximizes -RSS on the training set
- For reasons of emotional convenience (making things look familiar for our first pass), let's instead maximize $-\frac{1}{2}RSS$

$$-\frac{1}{2}RSS = -\frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  - If a particular function $\hat{f}(x_i)$ gives predictions that maximize $-RSS$ on the training set, then
  - $\hat{f}(x_i)$ also maximizes $-\frac{1}{2}RSS$ on the training set

Derivatives of -RSS tell us how to change our current predictions

- Each step in the gradient boosting process makes a small change to the predicted value $\hat{y}_i$ for each observation.
- How should we change the predicted values?
- The derivative of the negative RSS with respect to $\hat{y}_{i^*}$ (for a particular observation index i^*) tells us the rate of change of $-RSS$ if we make a small change to the predicted value $\hat{y}_{i^*}$ for that observation.

$$\begin{aligned}
\frac{\partial}{\partial \hat{y}_{i^*}} - \frac{1}{2}RSS &= \frac{\partial}{\partial \hat{y}_{i^*}}\left[-\frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\right] \\
&= \frac{\partial}{\partial \hat{y}_{i^*}} - \frac{1}{2}\left[(y_1 - \hat{y}_1)^2 + \cdots + (y_{i^*} - \hat{y}_{i^*})^2 + \cdots + (y_n - \hat{y}_n)^2\right] \\
&= -\frac{1}{2}2(y_{i^*} - \hat{y}_{i^*})(-1) \\
&= (y_{i^*} - \hat{y}_{i^*})
\end{aligned}$$

Interpretation: The (partial) derivative is equal to the residual.

- Suppose our current prediction is too small:
  - $y_{i^*} > \hat{y}_{i^*}$
  - $y_{i^*} - \hat{y}_{i^*} > 0$
  - Derivative is positive: We can increase -RSS by increasing $\hat{y}_{i^*}$
  - Larger difference between observed and predicted means larger derivative.
- Suppose our current prediction is too large:
  - $y_{i^*} < \hat{y}_{i^*}$
  - $y_{i^*} - \hat{y}_{i^*} < 0$
  - Derivative is negative: We can increase -RSS by decreasing $\hat{y}_{i^*}$
  - Larger difference between observed and predicted means larger (magnitude) derivative.

Estimation of the next component model

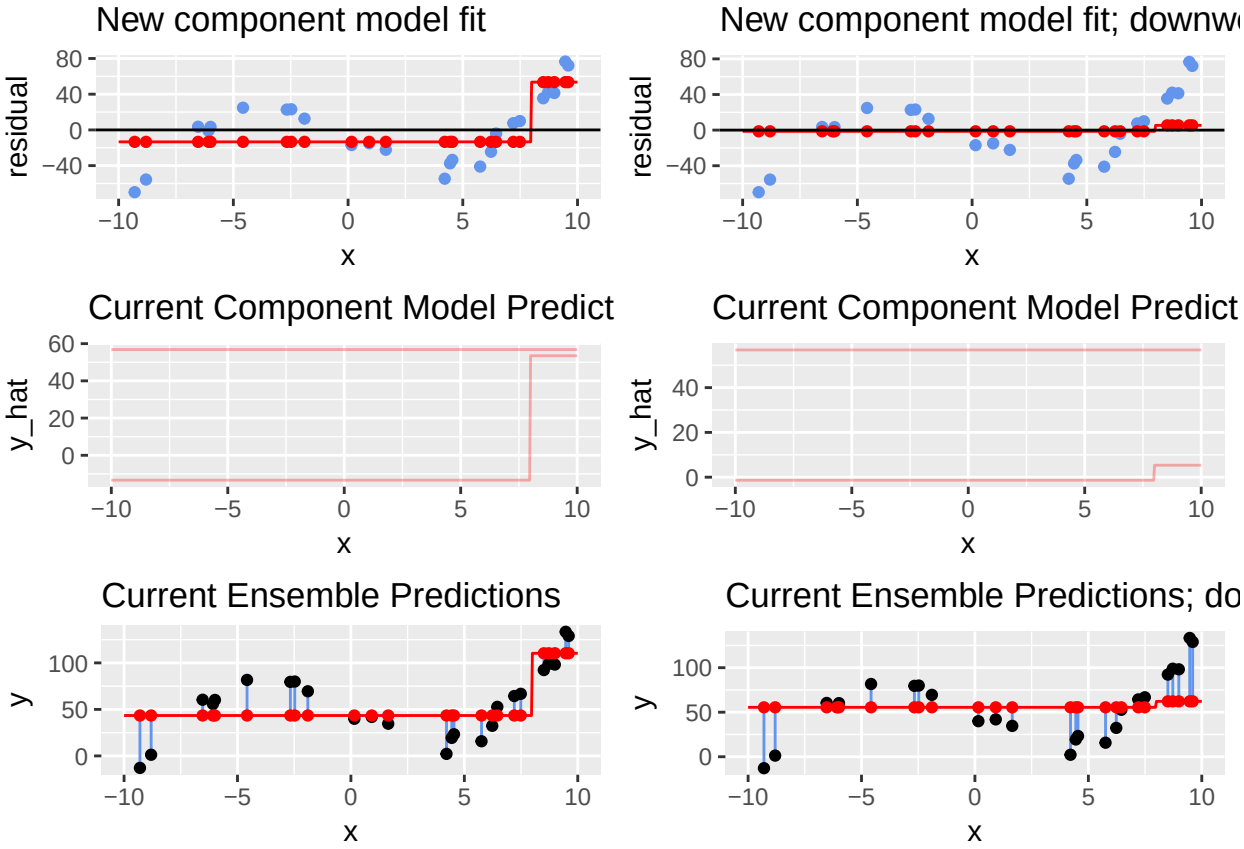
1. Calculate the **gradient vector** of our objective function with respect to the predicted values, evaluated at the current predictions from our ensemble:

$$\nabla_{\hat{y}}\left(-\frac{1}{2}RSS\right) = -\frac{1}{2}\left(\frac{\partial}{\partial\hat{y}_1}RSS, \ldots, \frac{\partial}{\partial\hat{y}_n}RSS\right) = (r_1, \ldots, r_n)$$

2. Fit a component model using the vector $(r_1, \ldots, r_n)$ as the response
3. Add the new component model to the ensemble.

Opportunities for Regularization/Preventing Overfitting

- **Learning Rate:** Multiply predictions from our new component model by a small weight like 0.01. Prevents us from immediately overfitting training data. Comparing step 1 with learning rate 1 and learning rate 0.1:



- **Number of boosting iterations:** The more boosting iterations we run, the more we run the risk of overfitting.
- **Minimum Reduction in RSS:** How big does a gain from a split need to be, in order to make that split?
- **Tree Depth:** Above we fit "decision stumps": 1 split only. Allowing deeper trees allows more flexible models.
- **Train on Fewer Observations:** Each tree trained using a subset of training set observations.
- **Train on Fewer Features:** Each component model trained using a subset of available explanatory variables.

Miscellaneous Notes:

- The scaling factor of 1/2 is arbitrary - especially if we use a small learning rate.
- At some point we will need to deal with a negative sign: either to get an objective function to maximize, or to indicate the direction to move to minimize RSS.

Estimation with xgboost ("eXtreme Gradient Boosting")

- Data scientists have gotten better at catchy names since the days of Type I/Type II errors.
- One of several commonly used implementations of gradient boosting. Written in C, interfaces to other languages like python
- Second-most-commonly used option is lightgbm
- Estimation can be done via the train function in the caret package.

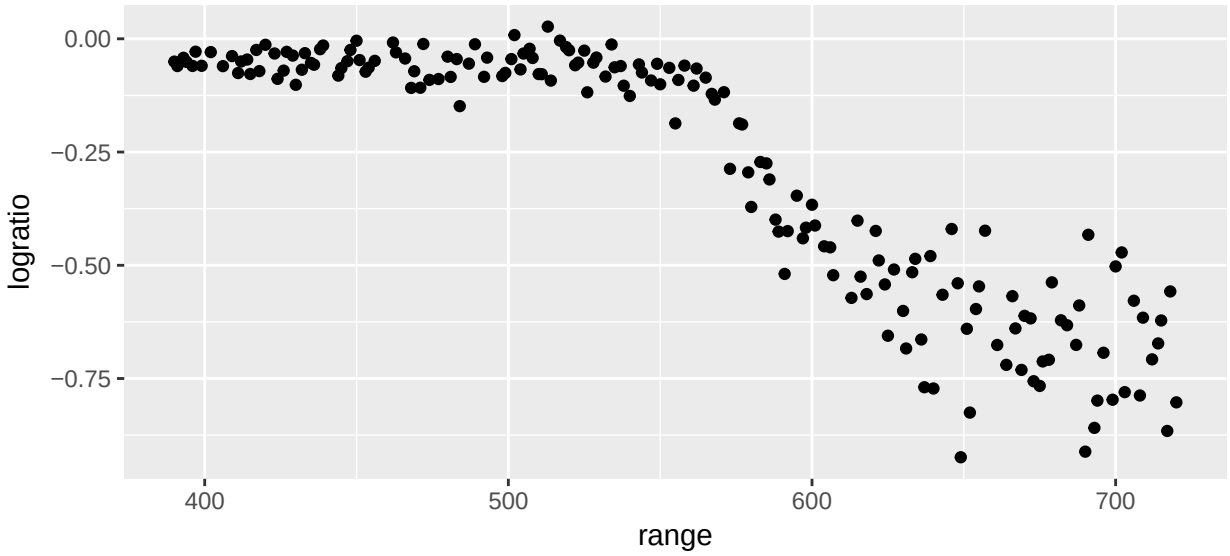
Let's look at our favorite lidar data set:

```
library(readr)
lidar <- read_table2("http://www.evanlray.com/data/all-of-nonparametric-stat/lidar.dat")
```

```
## Parsed with column specification:
## cols(
##   range = col_integer(),
##   logratio = col_double()
## )
```

```
tt_split <- caret::createDataPartition(lidar$logratio, p = 0.8)
lidar_train <- lidar %>% slice(tt_split[[1]])
lidar_test <- lidar %>% slice(-tt_split[[1]])

ggplot(data = lidar_train, mapping = aes(x = range, y = logratio)) +
  geom_point()
```




```
library(caret)
xgb_fit <- train(
  logratio ~ range,
  data = lidar_train,
  method = "xgbTree",
  trControl = trainControl(method = "cv", number = 10, returnResamp = "all"),
  tuneGrid = expand.grid(
    nrounds = c(10, 50, 100),
    eta = 0.3, # learning rate; 0.3 is the default
    gamma = 0, # minimum loss reduction to make a split; 0 is the default
    max_depth = 1:5, # how deep are our trees?
    subsample = c(0.8, 1), # proportion of observations to use in growing each tree
    colsample_bytree = 1, # proportion of explanatory variables used in each tree
    min_child_weight = 1 # think of this as how many observations must be in each leaf node
  )
)
```

```
xgb_fit$results %>% select(nrounds, max_depth, subsample, RMSE)
```

```
##    nrounds max_depth subsample       RMSE
## 1       10         1       0.8 0.08053794
## 4       10         1       1.0 0.08132399
## 7       10         2       0.8 0.08405485
## 10      10         2       1.0 0.08342368
## 13      10         3       0.8 0.08232797
## 16      10         3       1.0 0.08722215
## 19      10         4       0.8 0.08792820
## 22      10         4       1.0 0.08608794
## 25      10         5       0.8 0.08926086
## 28      10         5       1.0 0.08928720
## 2       50         1       0.8 0.08238081
## 5       50         1       1.0 0.08267746
## 8       50         2       0.8 0.09365501
## 11      50         2       1.0 0.09422426
## 14      50         3       0.8 0.10030162
## 17      50         3       1.0 0.10126625
## 20      50         4       0.8 0.10810732
## 23      50         4       1.0 0.10385971
## 26      50         5       0.8 0.11049344
## 29      50         5       1.0 0.10715467
## 3      100         1       0.8 0.08481607
## 6      100         1       1.0 0.08458801
## 9      100         2       0.8 0.10224549
## 12     100         2       1.0 0.10258051
## 15     100         3       0.8 0.10655722
## 18     100         3       1.0 0.10661587
## 21     100         4       0.8 0.11329340
## 24     100         4       1.0 0.10920066
## 27     100         5       0.8 0.11512836
## 30     100         5       1.0 0.11186667
```

Looks like we may be overfitting; our best RMSE is with the lowest values of max depth and nrounds. Let's try a lower learning rate. Also, subsample wasn't helpful. Let's just stick with subsample = 1.

```
library(caret)
xgb_fit <- train(
  logratio ~ range,
  data = lidar_train,
  method = "xgbTree",
  trControl = trainControl(method = "cv", number = 10, returnResamp = "all"),
  tuneGrid = expand.grid(
    nrounds = c(5, 10, 20, 30, 40),
    eta = c(0.1, 0.2, 0.3), # learning rate; 0.3 is the default
    gamma = 0, # minimum loss reduction to make a split; 0 is the default
    max_depth = 1:2, # how deep are our trees?
    subsample = 1, # proportion of observations to use in growing each tree
    colsample_bytree = 1, # proportion of explanatory variables used in each tree
    min_child_weight = 1 # think of this as how many observations must be in each leaf node
  )
)

xgb_fit$results %>% filter(RMSE == min(RMSE))
```

```
##   eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 1 0.2         1     0                1                1         1      20
##         RMSE  Rsquared        MAE     RMSESD RsquaredSD      MAESD
## 1 0.07816437 0.9247295 0.05410836 0.01874173 0.02989515 0.01159727
```

The best tuning parameter values were the middle of the ranges of values we tried (or at the edge of possible values, in the case of max_depth); seems OK.

Let's look at the predictions:

```
lidar_test <- lidar_test %>%
  mutate(
    logratio_hat = predict(xgb_fit, lidar_test)
  )

ggplot() +
  geom_point(data = lidar_train, mapping = aes(x = range, y = logratio)) +
  geom_point(data = lidar_test, mapping = aes(x = range, y = logratio), color = "orange") +
  geom_line(data = lidar_test, mapping = aes(x = range, y = logratio_hat), color = "orange")
```