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 (bootstrap sample)
- Each tree uses a random subset of features in searching for each split.
- The trees are all estimated separately, then predictions are combined later.
- For example, for regression:

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

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

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

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?

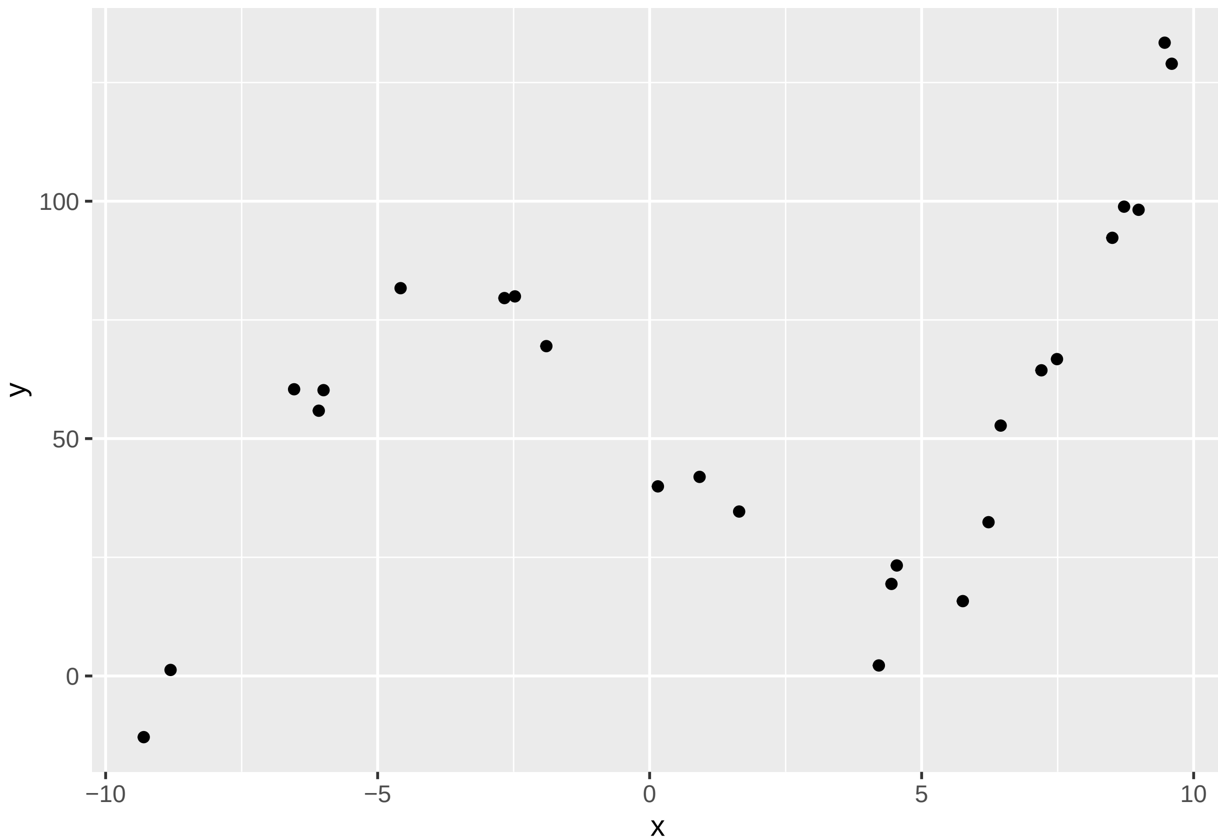
- New component models are specifically different from what's already in the ensemble!

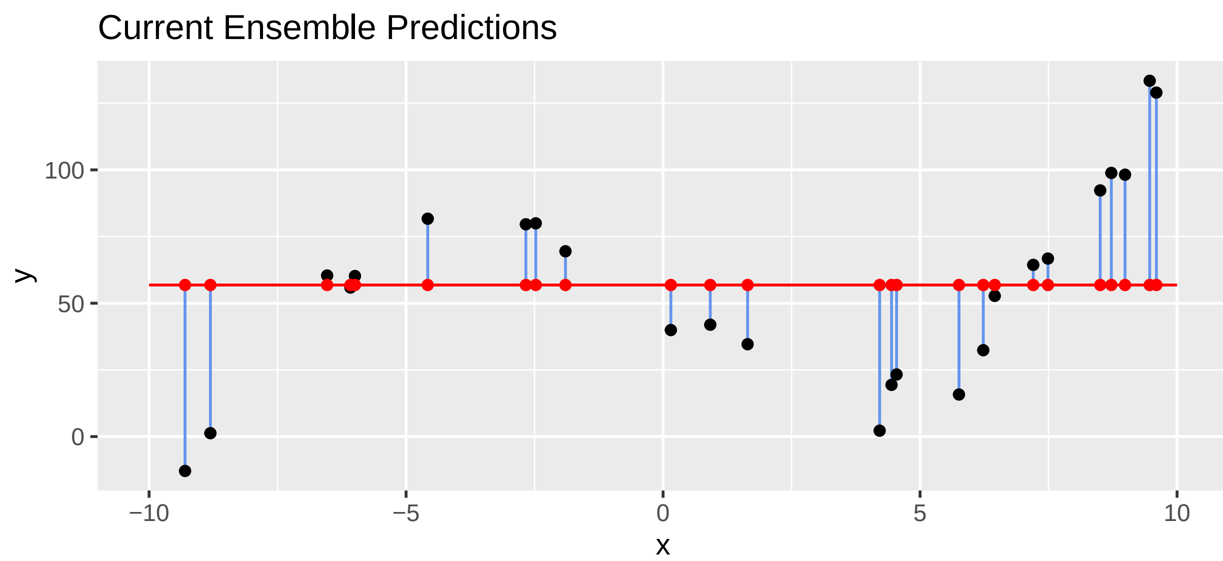
A Specific Example: Gradient Tree Boosting

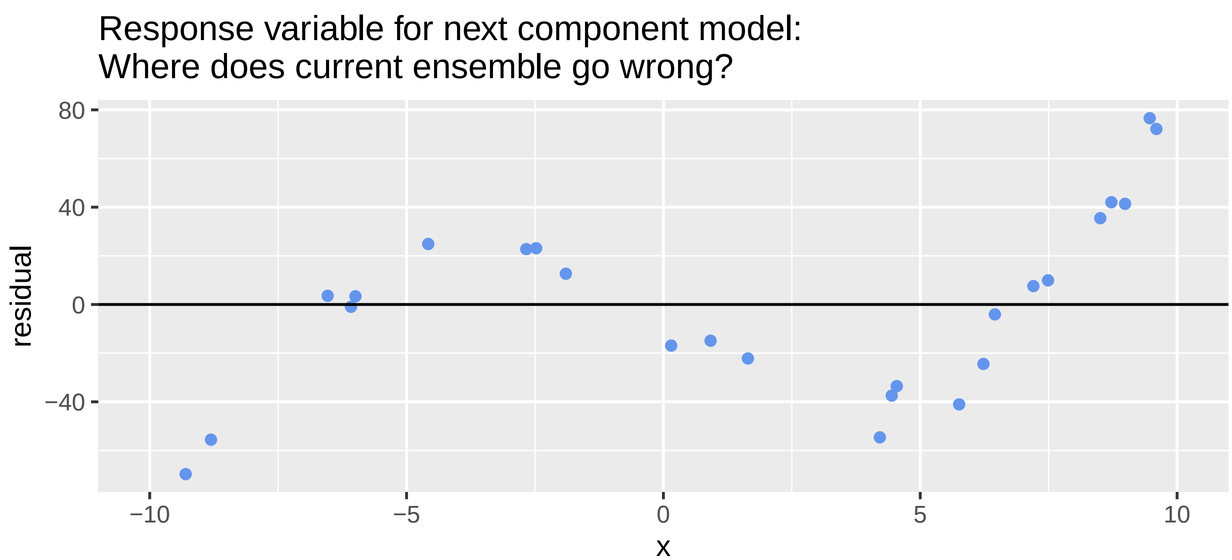
Let's start with building some intuition for the method, and define it more carefully later.

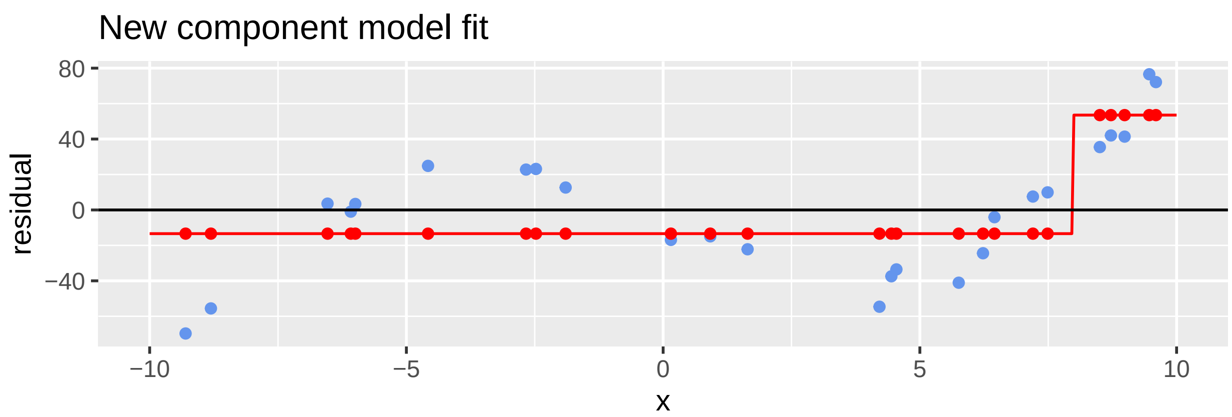
In this example, our component models will be "stumps": trees with only one split.

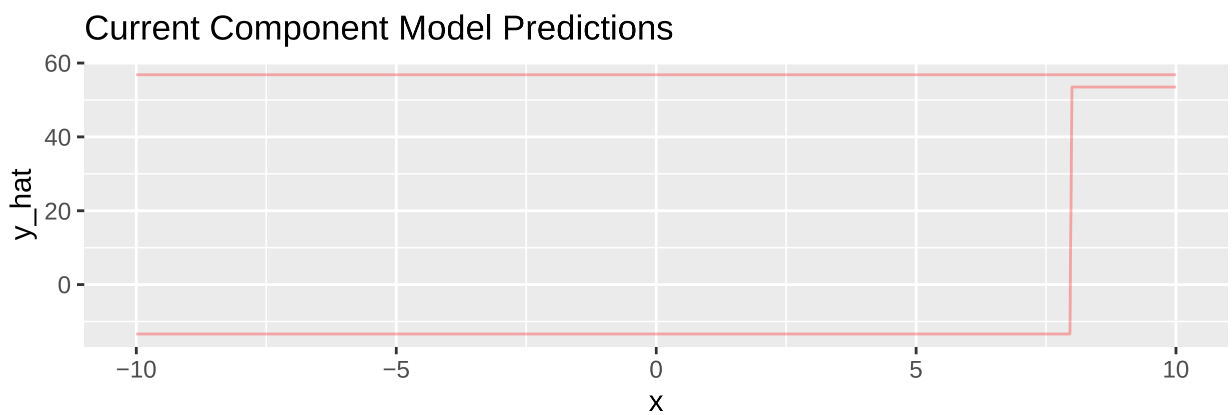
Here's a made up regression problem, and an initial prediction for each observation, given by the sample mean for the response variable.

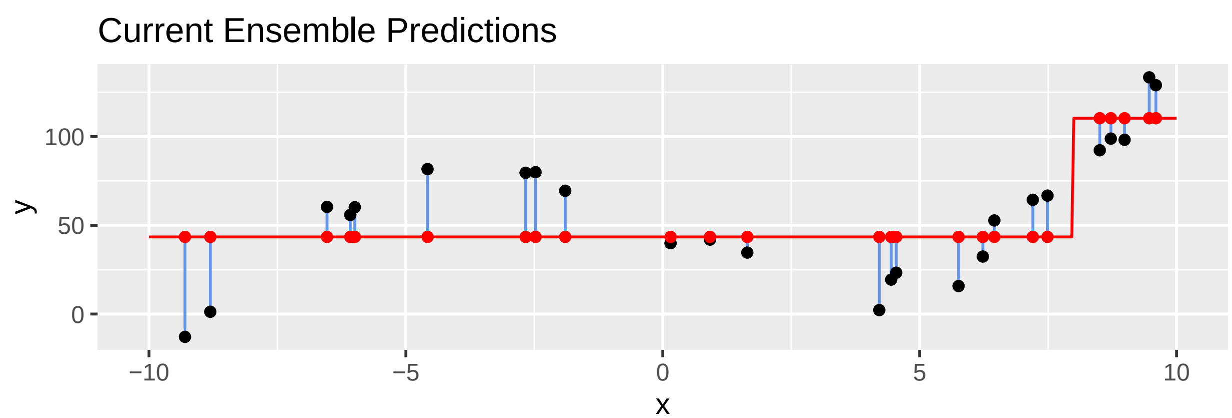


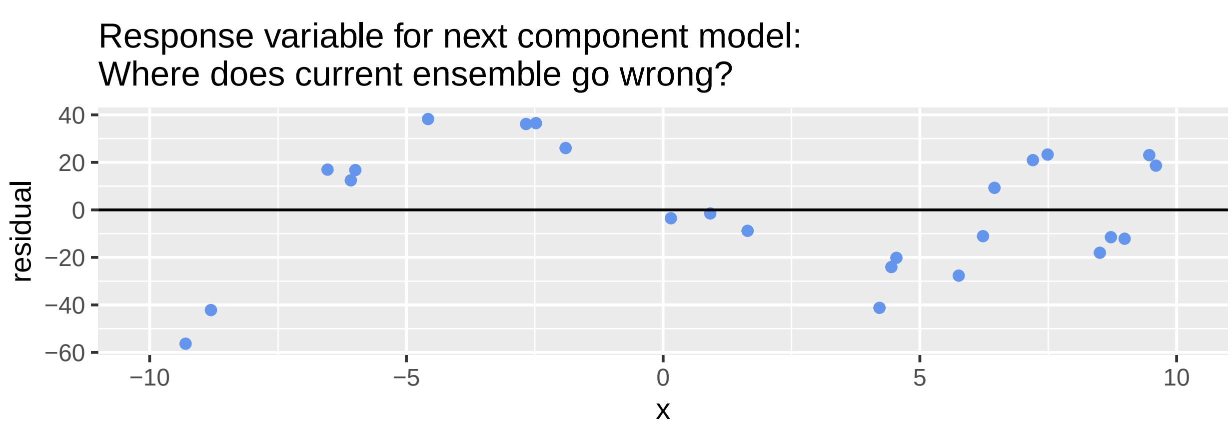

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

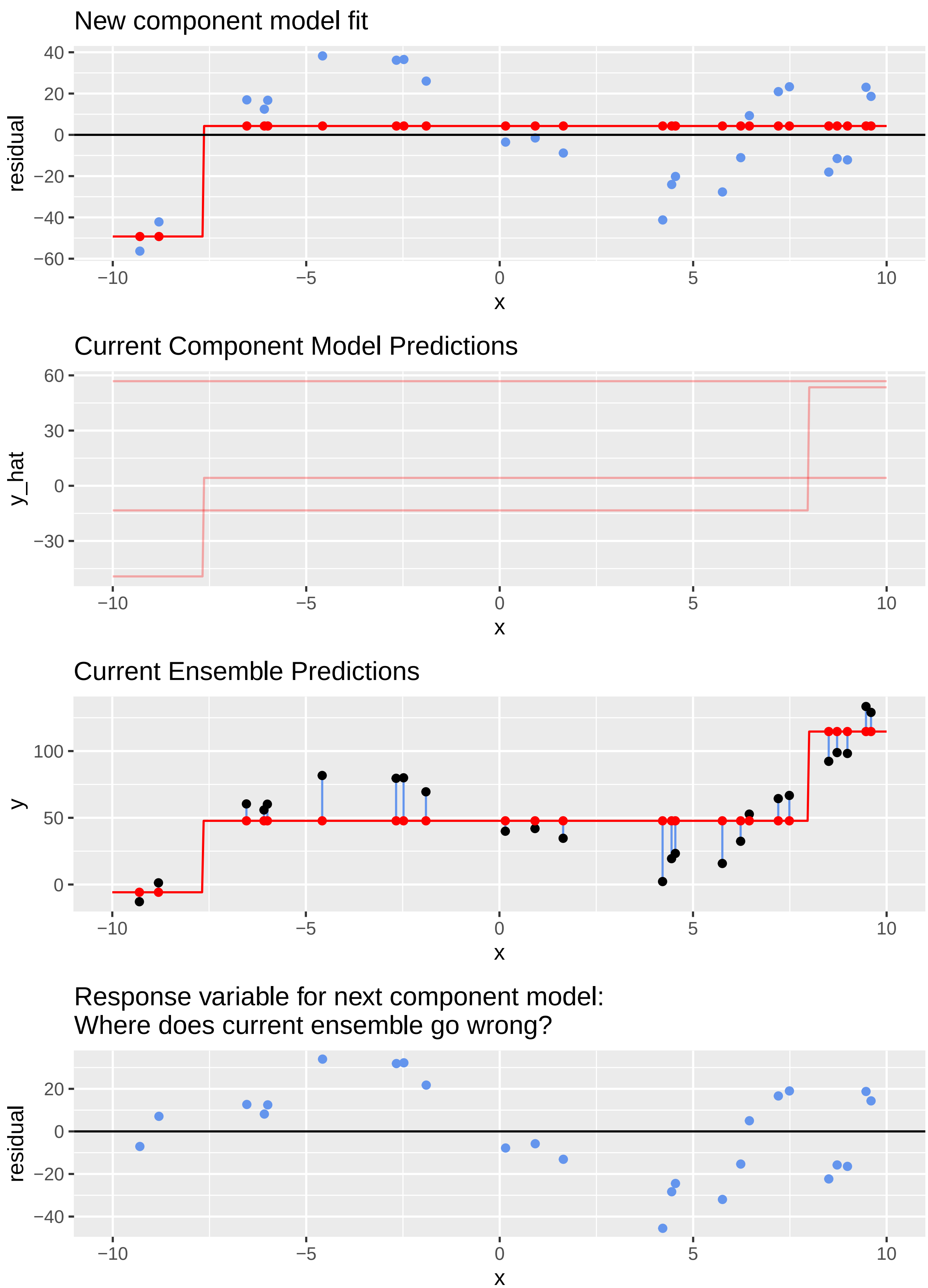

Response variable for next component model:
Where does current ensemble go wrong?
residual
80
40
0
−40
−10
−5
0
5
10
x

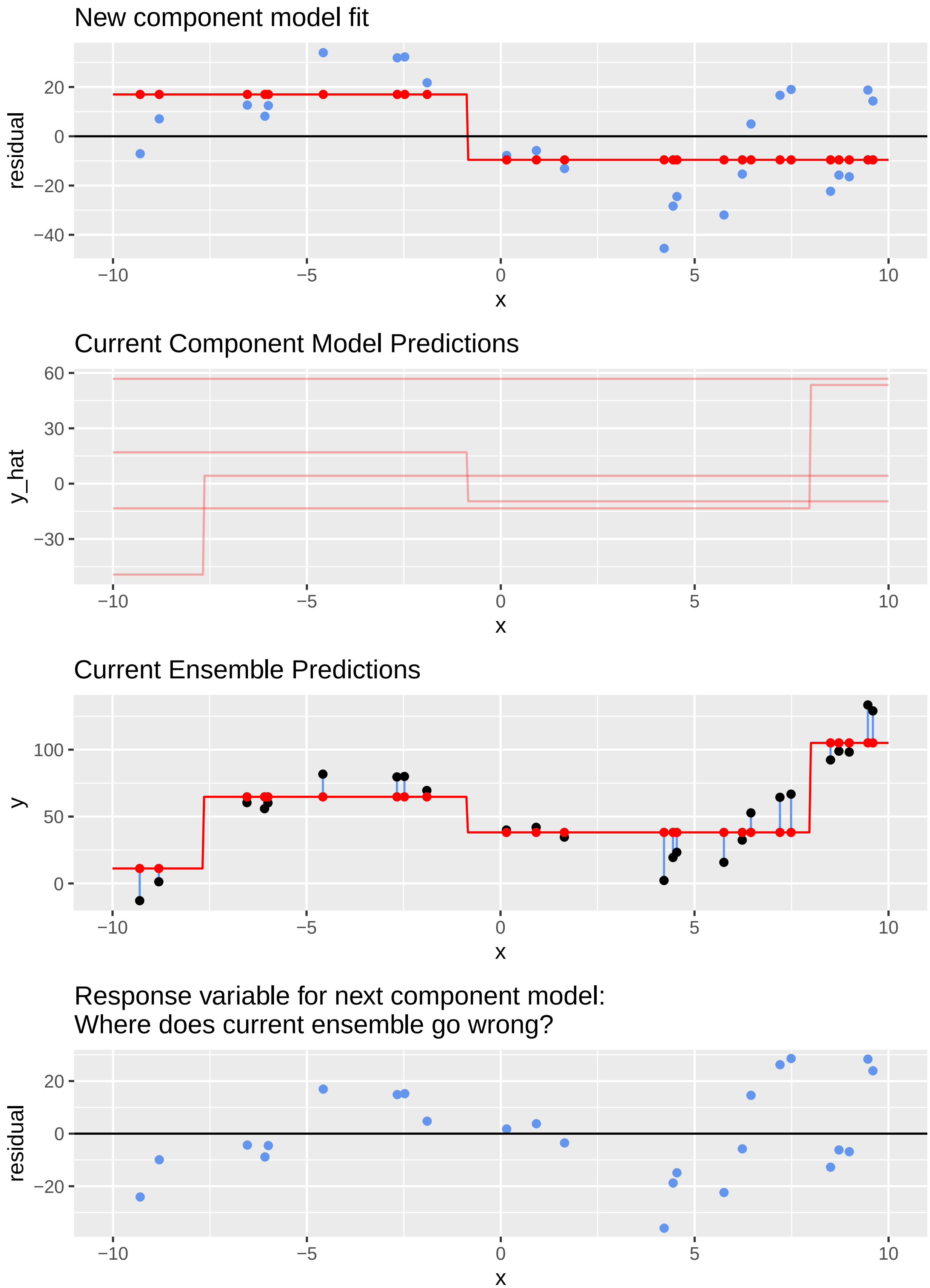

New component model fit
residual
80
40
0
−40
−10
−5
0
5
10
x

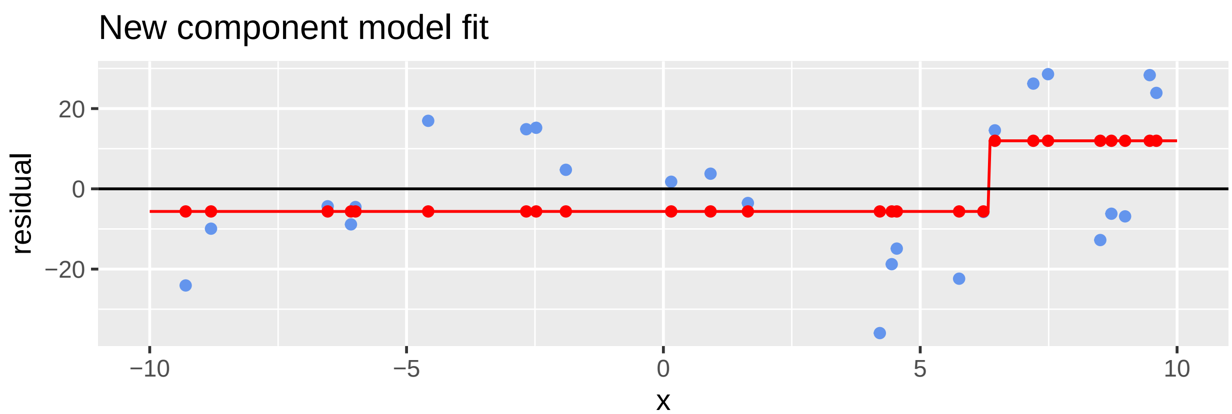

Current Component Model Predictions
y_hat
60
40
20
0
−10
−5
0
5
10
x

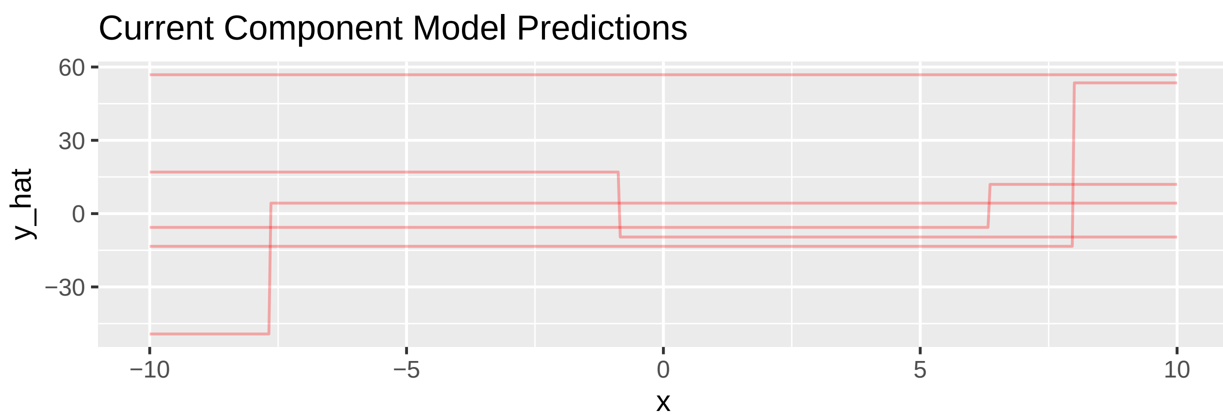

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

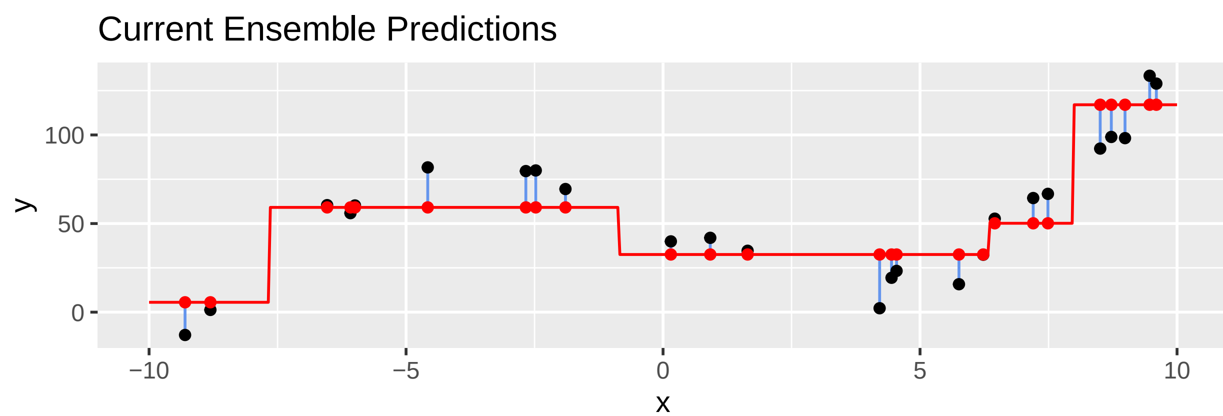

Response variable for next component model:
Where does current ensemble go wrong?
residual
40
20
0
−20
−40
−60
−10
−5
0
5
10
x

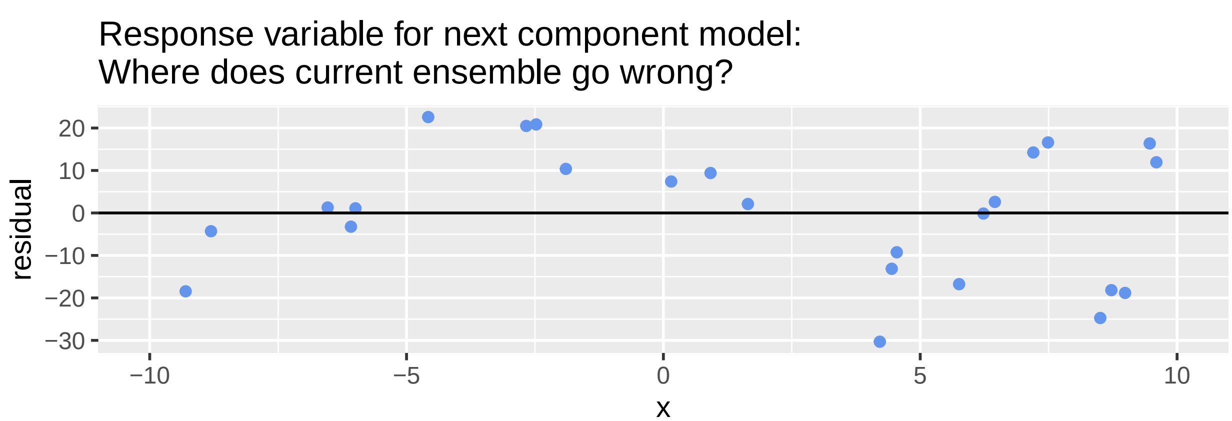

New component model fit
residual
Current Component Model Predictions
y_hat
Current Ensemble Predictions
y
Response variable for next component model:
Where does current ensemble go wrong?
residual
x

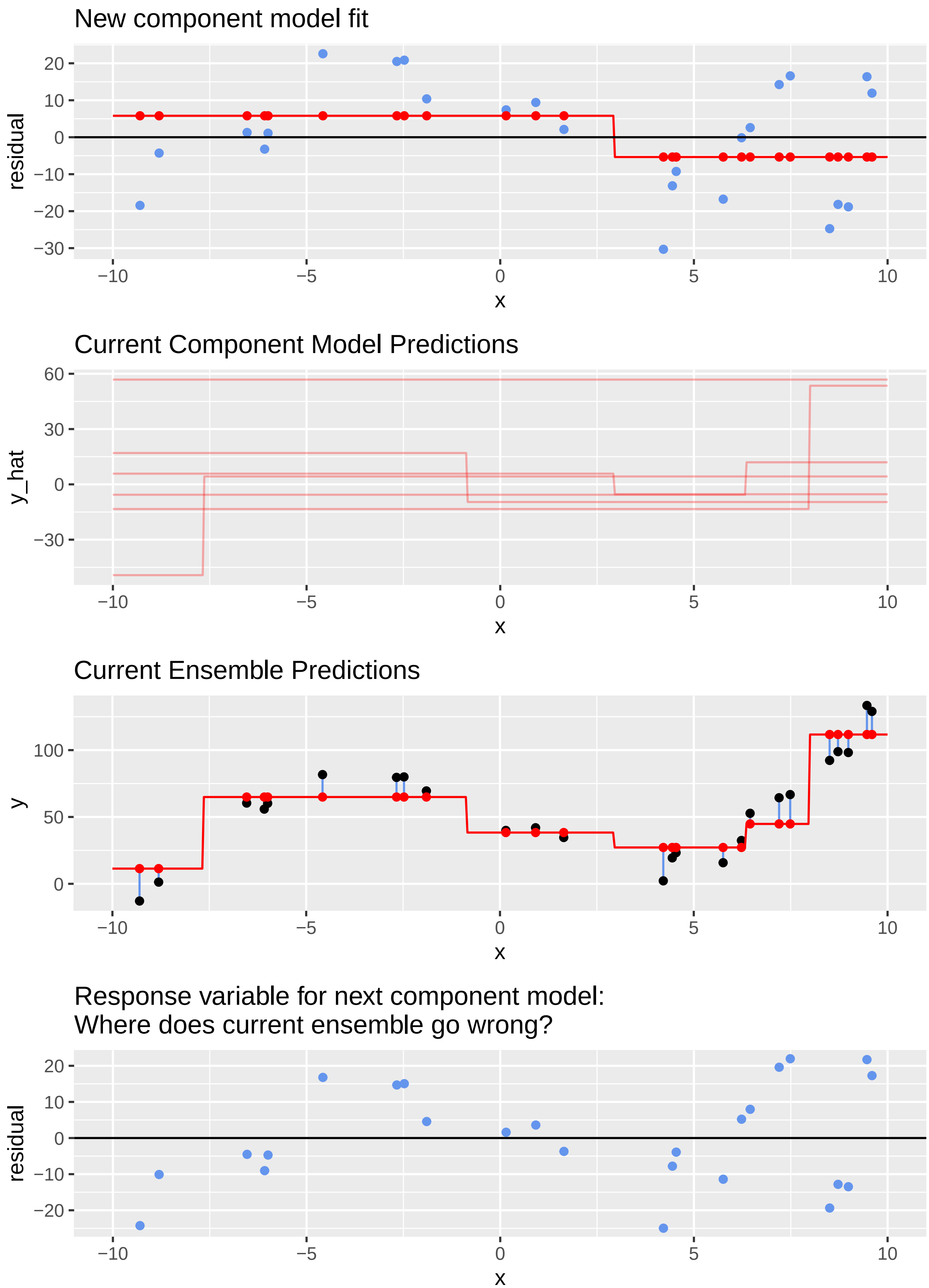

New component model fit
residual
20
0
−20
−40
−10
−5
0
5
10
x
Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x
Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x
Response variable for next component model:
Where does current ensemble go wrong?
residual
20
0
−20
−10
−5
0
5
10
x


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

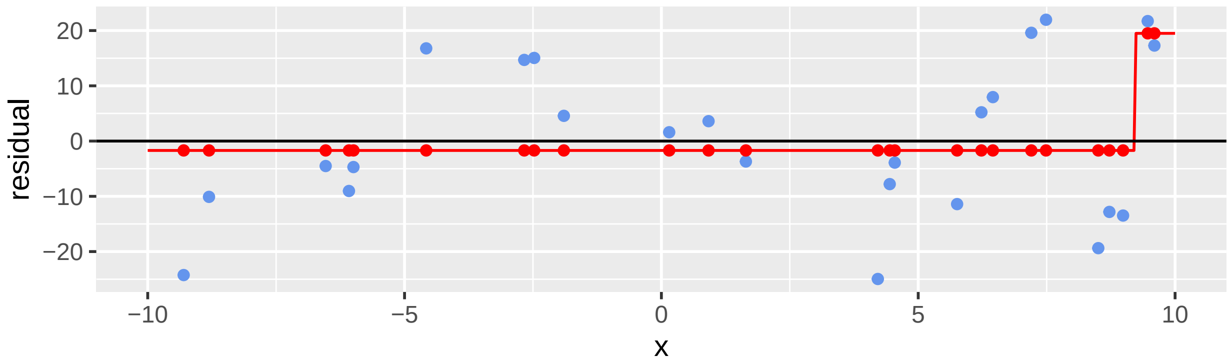

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x


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

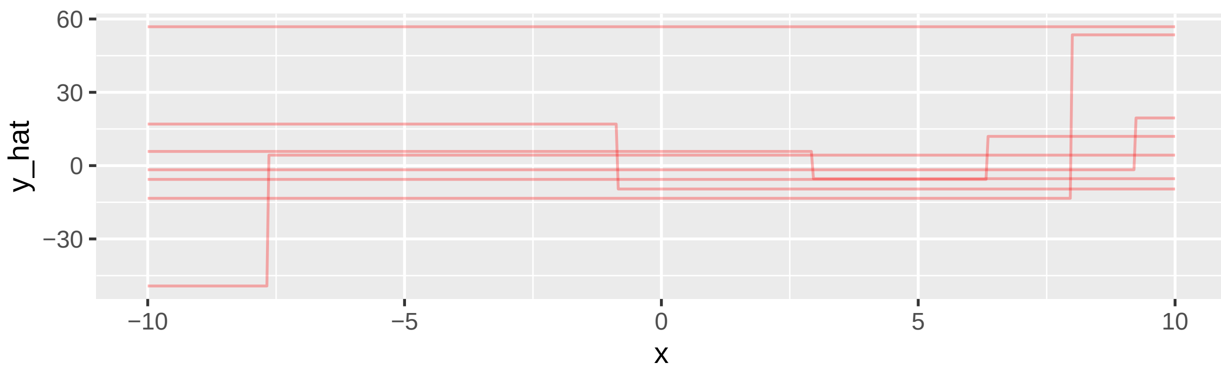

Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
-10
-20
-30
−10
−5
0
5
10
x


New component model fit
residual
x
Current Component Model Predictions
y_hat
x
Current Ensemble Predictions
y
x
Response variable for next component model:
Where does current ensemble go wrong?
residual
x

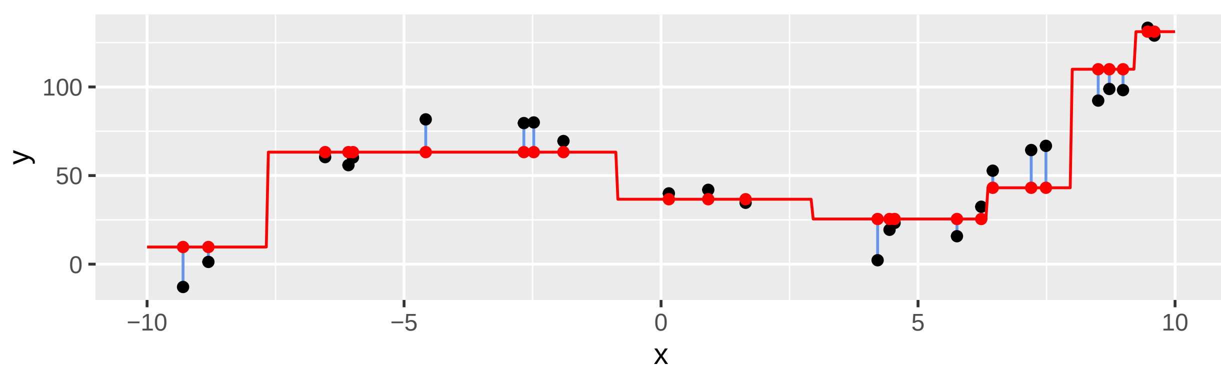
New component model fit



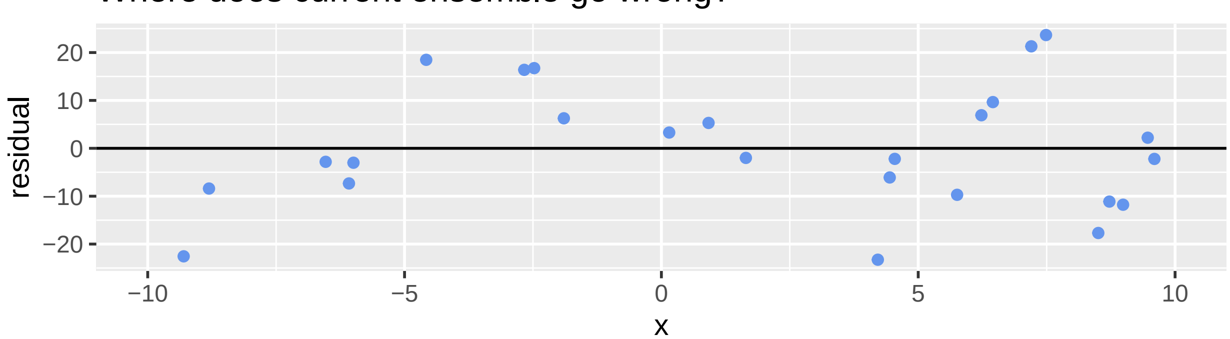
Current Component Model Predictions

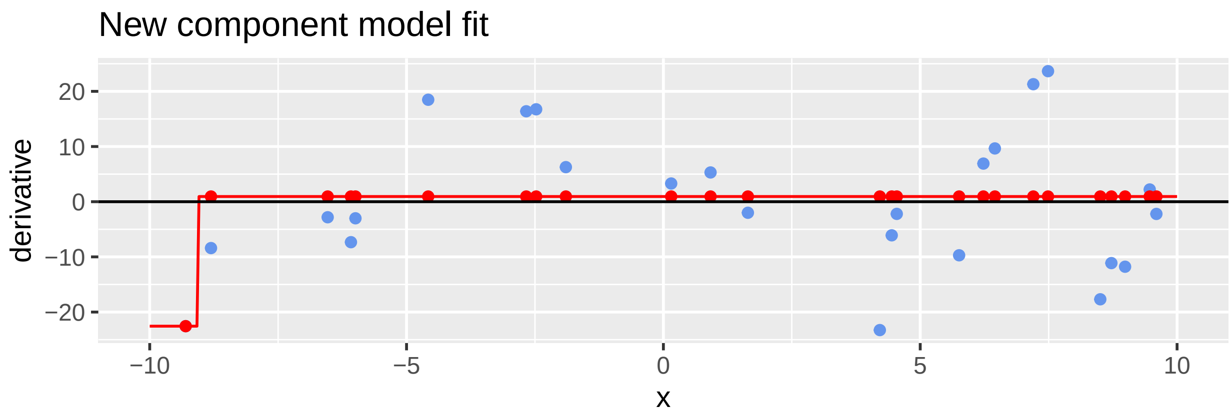


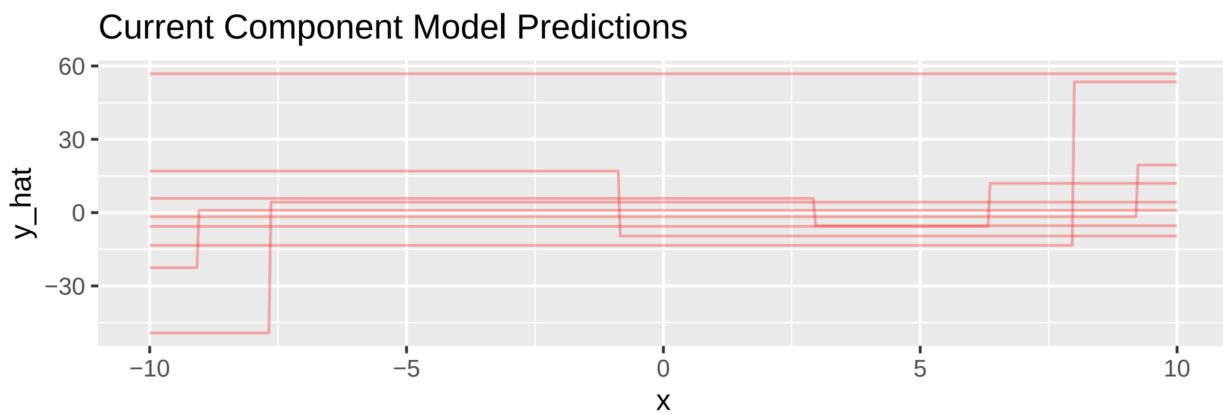
Current Ensemble Predictions

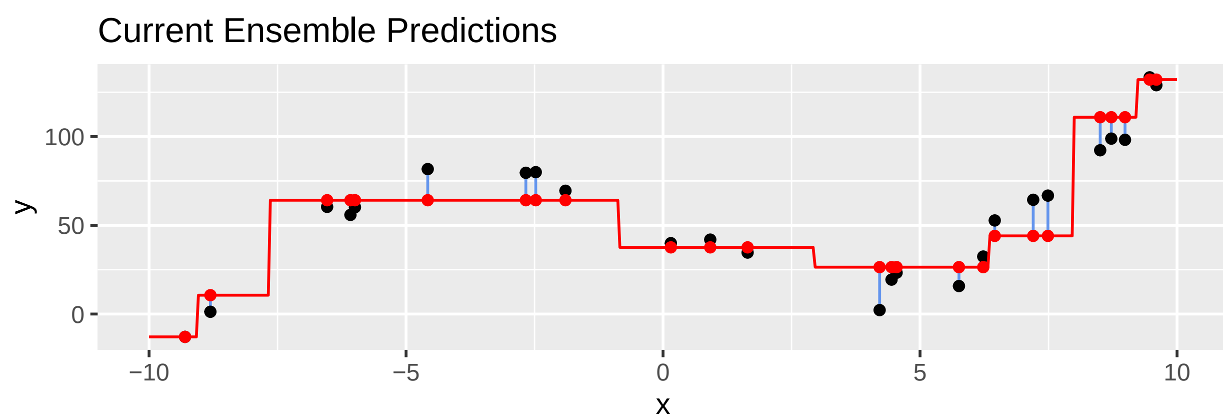


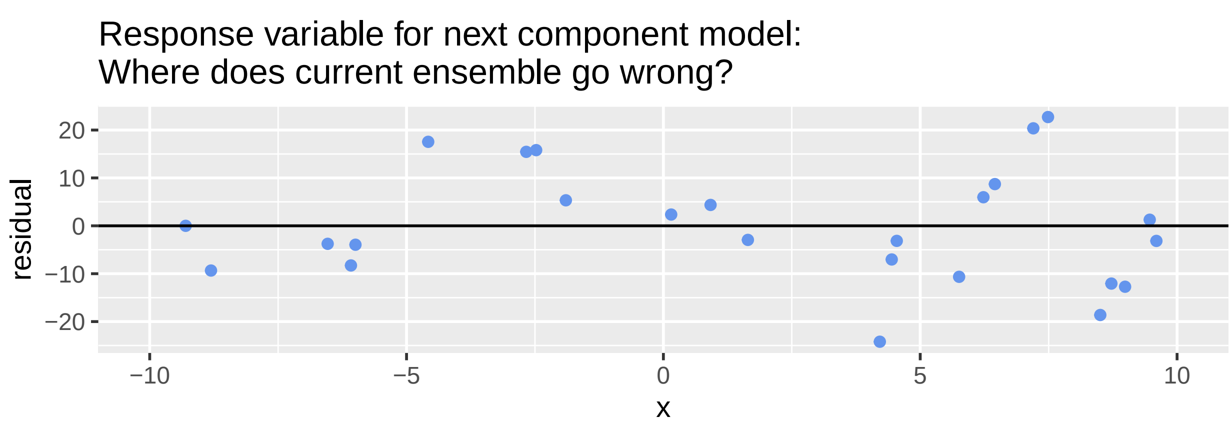
Response variable for next component model:
Where does current ensemble go wrong?

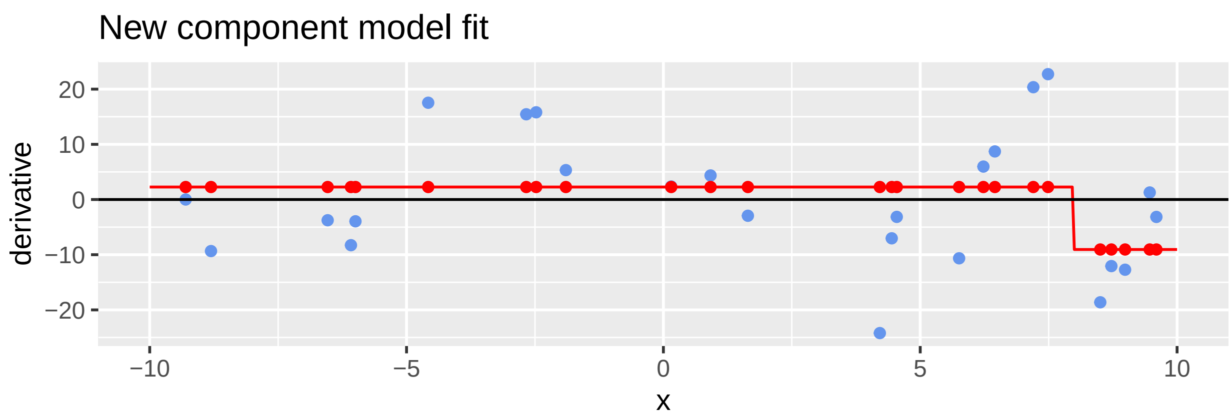


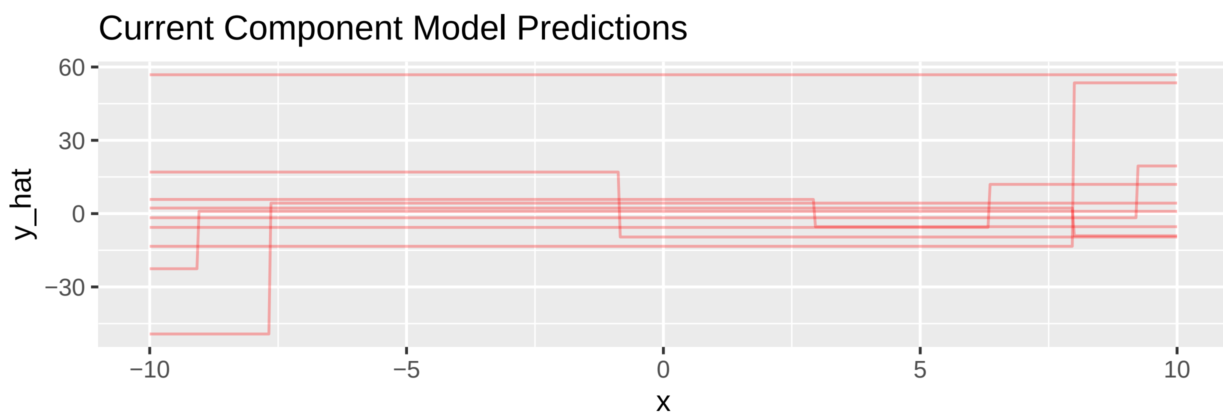

New component model fit
derivative
20
10
0
−10
−20
−10
−5
0
5
10
x

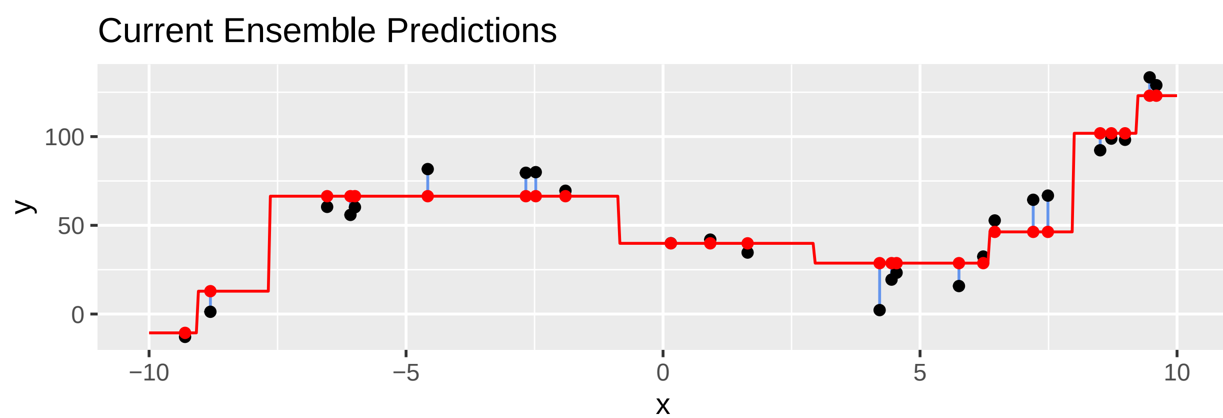

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x

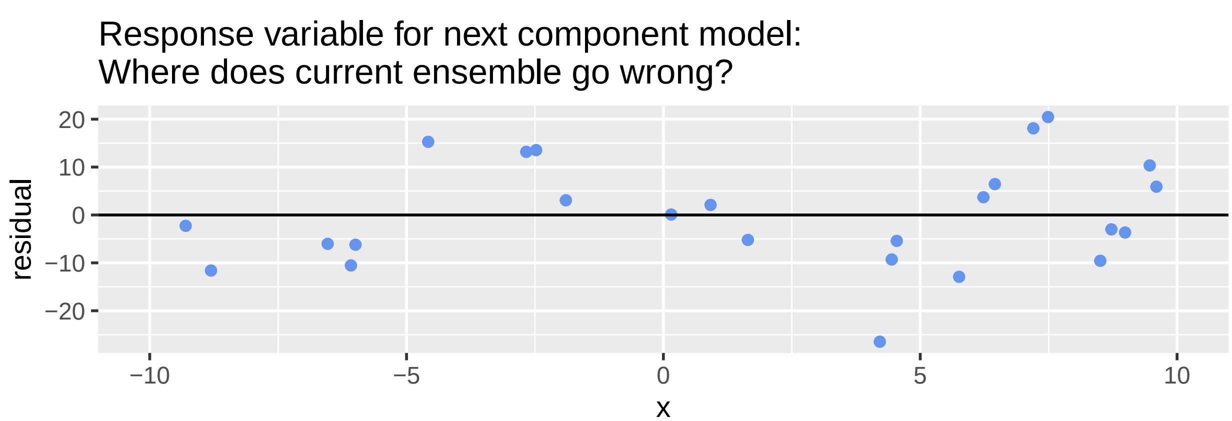

Current Ensemble Predictions
y
100
50
0
−10
−5
0
5
10
x

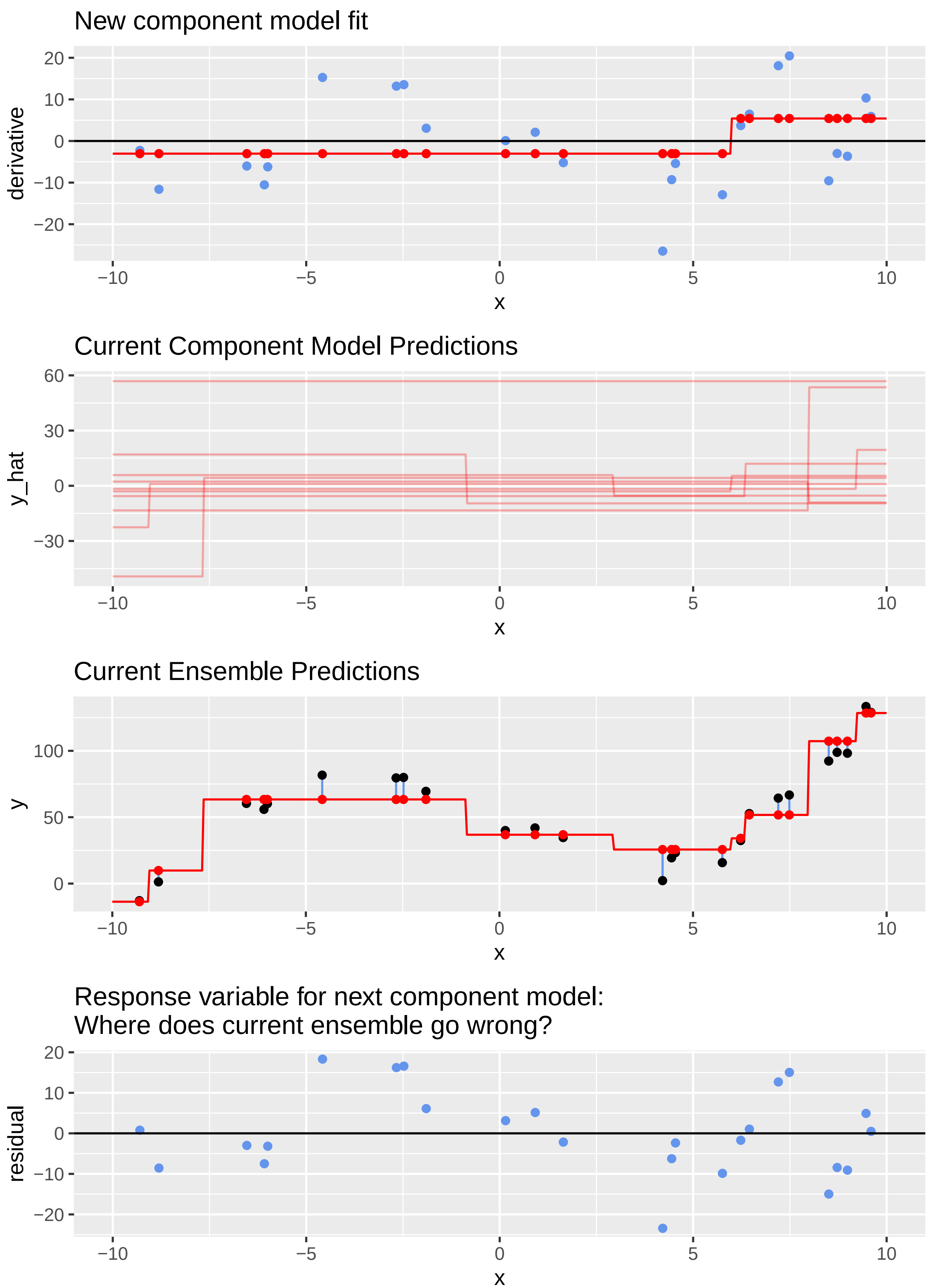

Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−10
−5
0
5
10
x


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

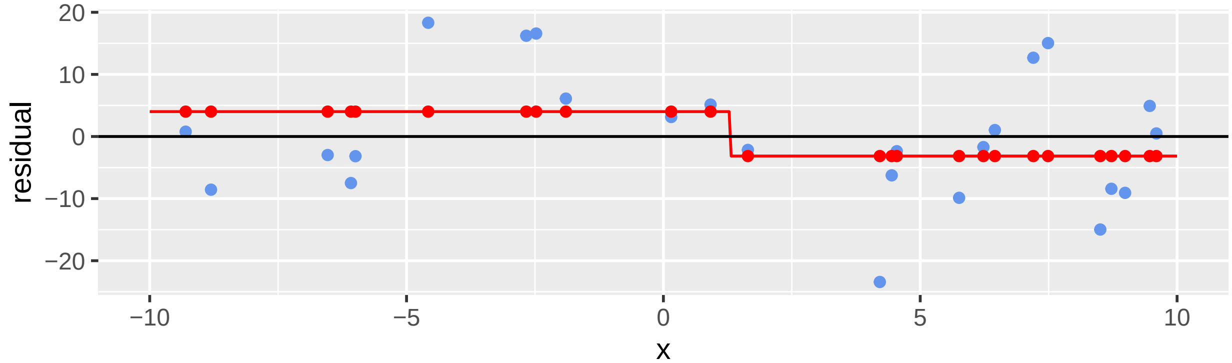

Current Component Model Predictions
y_hat
60
30
0
−30
−10
−5
0
5
10
x


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

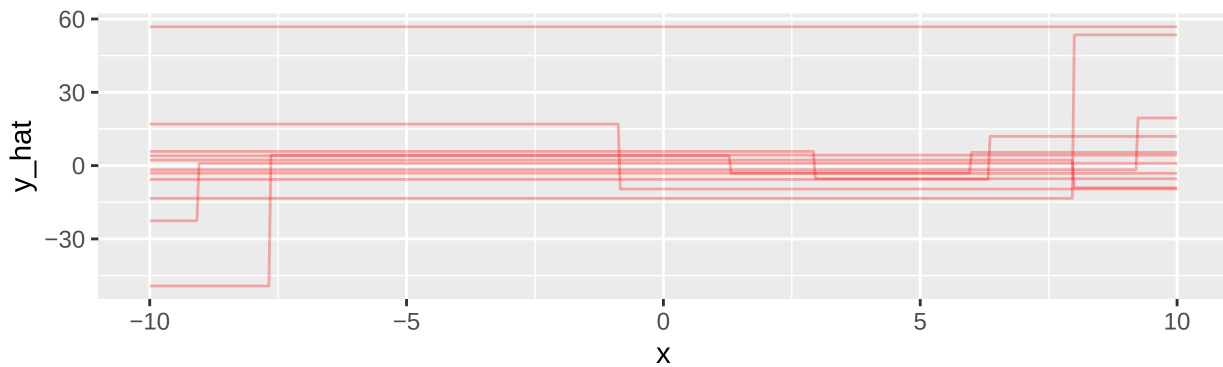

Response variable for next component model:
Where does current ensemble go wrong?
residual
20
10
0
−10
−20
−10
−5
0
5
10
x


New component model fit
derivative
x
Current Component Model Predictions
y_hat
x
Current Ensemble Predictions
y
x
Response variable for next component model:
Where does current ensemble go wrong?
residual
x

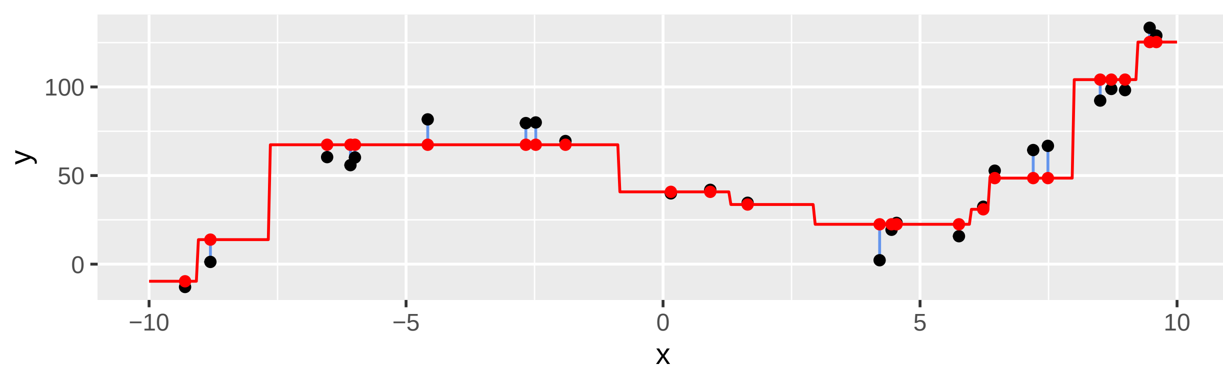
New component model fit



Current Component Model Predictions



Current Ensemble Predictions



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

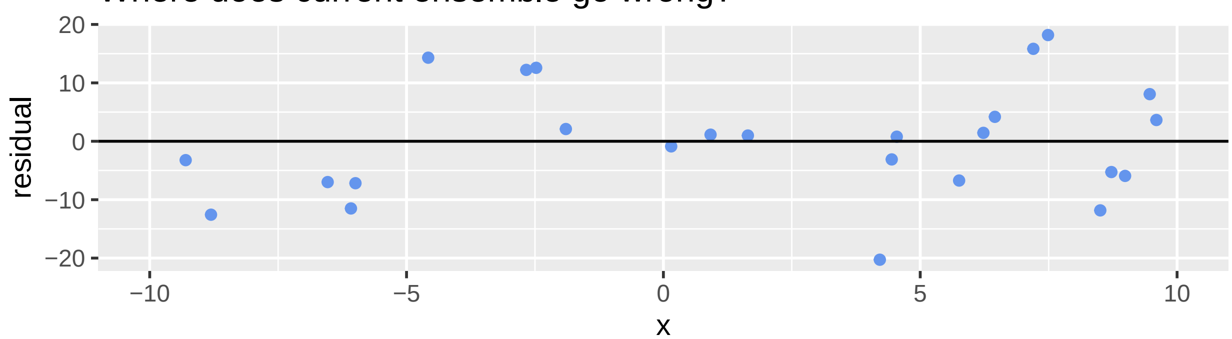
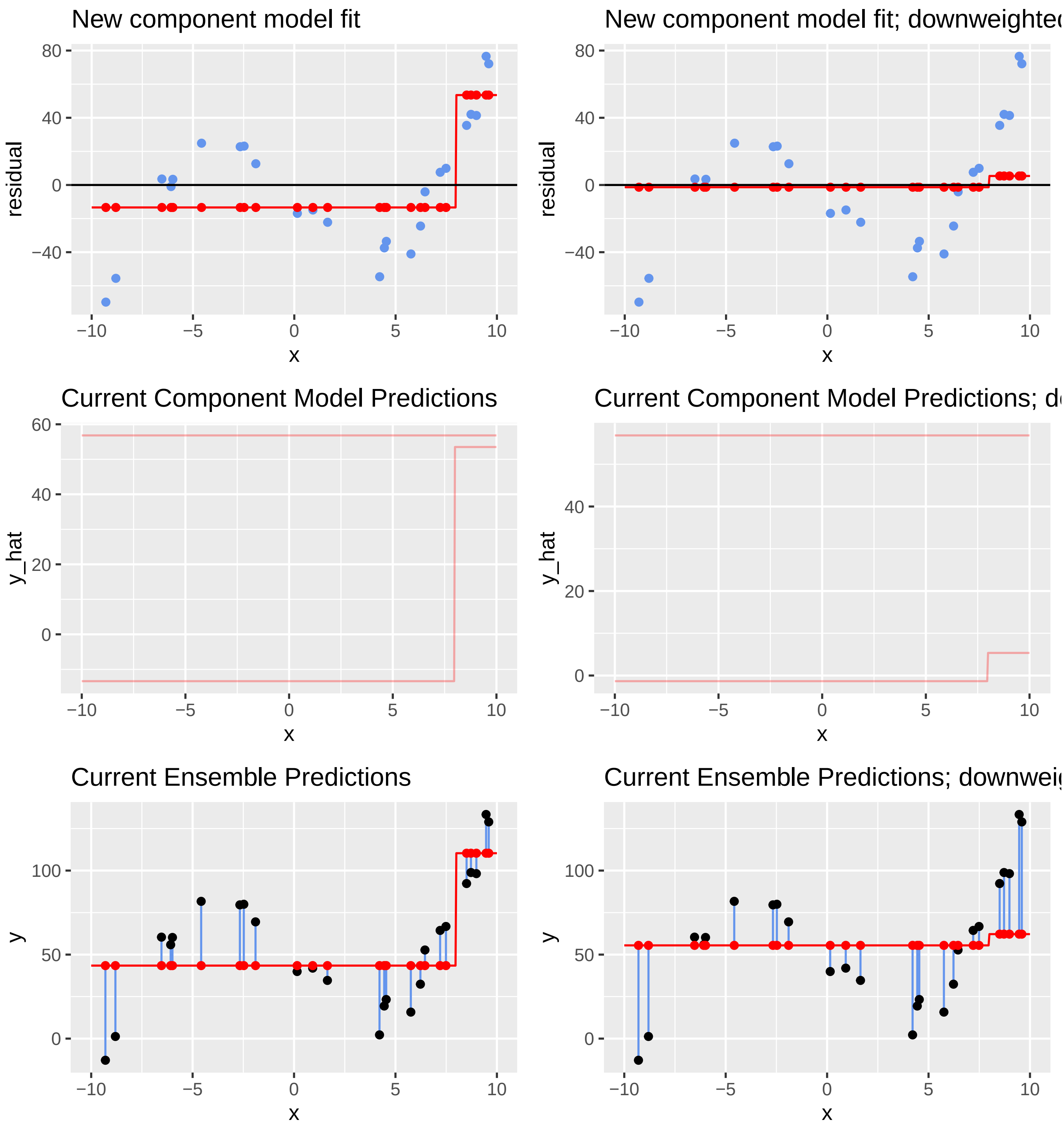


Illustration of Learning Rate

- **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:



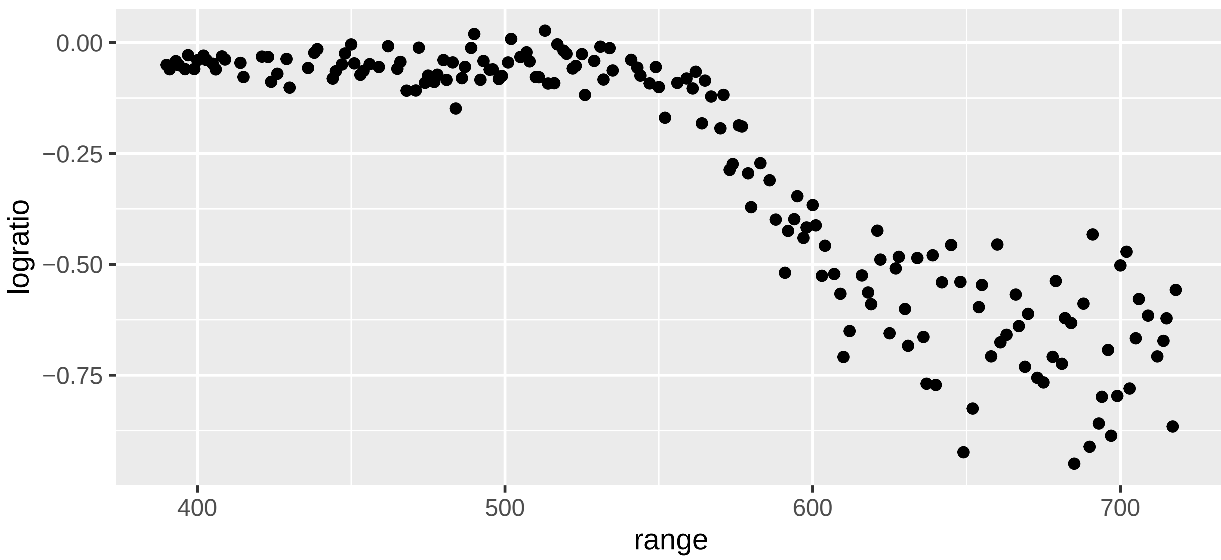
Estimation with xgboost ("eXtreme Gradient Boosting")

- 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 R and python
- Estimation can be done via the train function in the caret package.

Let's look at the lidar data set:

```
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 0.08730318
## 4       10         1       1.0 0.08668356
## 7       10         2       0.8 0.08601407
## 10      10         2       1.0 0.08837704
## 13      10         3       0.8 0.08885059
## 16      10         3       1.0 0.09160769
```

```
## 19      10         4       0.8 0.09198080
## 22      10         4       1.0 0.09411309
## 25      10         5       0.8 0.09230264
## 28      10         5       1.0 0.09614579
## 2       50         1       0.8 0.09002320
## 5       50         1       1.0 0.08884638
## 8       50         2       0.8 0.09982263
## 11      50         2       1.0 0.10238349
## 14      50         3       0.8 0.10497632
## 17      50         3       1.0 0.10824797
## 20      50         4       0.8 0.11068940
## 23      50         4       1.0 0.11058551
## 26      50         5       0.8 0.11197533
## 29      50         5       1.0 0.11367366
## 3      100         1       0.8 0.09282210
## 6      100         1       1.0 0.09134740
## 9      100         2       0.8 0.10732604
## 12     100         2       1.0 0.10824223
## 15     100         3       0.8 0.10952251
## 18     100         3       1.0 0.11248027
## 21     100         4       0.8 0.11385201
## 24     100         4       1.0 0.11522892
## 27     100         5       0.8 0.11633177
## 30     100         5       1.0 0.11669585
```

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.

```r
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 0.08028648 0.9301506 0.05703936 0.02994344 0.03879195 0.0192988
```

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:

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

```r
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")
```