Regression deals with CONTINUOUS responses (outputs)

- In layman’s terms, a continuous variable is a decimal number.
- 1.0, 2.0, 3.0, 3.2, 3.2221, 5.01, …
- Think of a continuous response as a floating point number.
- Or more formally, there are an infinite number of real values within any interval.

Let’s work on a concrete, yet simple, example

- We said that all relationships learned from data are approximate, $y \approx f(\mathbf{x})$.

- However, let’s create an example where we know **ground truth**.

- We know the TRUE relationship between a NOISE-FREE signal and a single input variable, x.

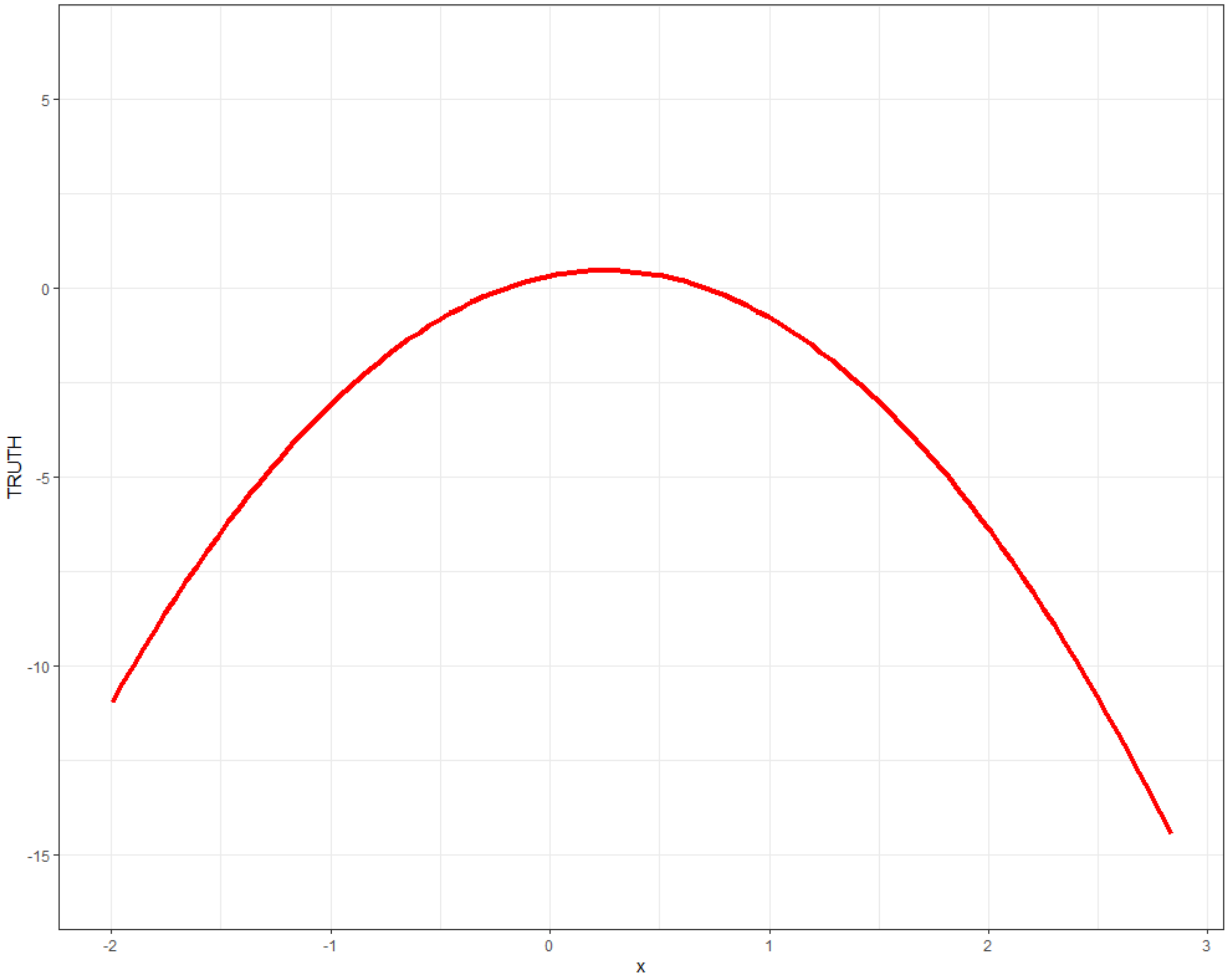
The **TRUE** relationship will be assumed to be a quadratic relationship

$$\text{TRUTH} = \beta_{*0} + \beta_{*1}x + \beta_{*2}x^2$$

Set the coefficients or parameters equal to:

$$\beta_{*0} = 0.33, \beta_{*1} = 1.15, \text{and } \beta_{*2} = -2.25$$

The true quadratic trend is a parabola!



We never observe TRUTH…when we collect data, we record <u>NOISY</u> observations

- The TRUE or NOISE-FREE signal is HIDDEN from us.

- We “see” a corrupted signal.

We never observe TRUTH…when we collect data, we record **NOISY** observations

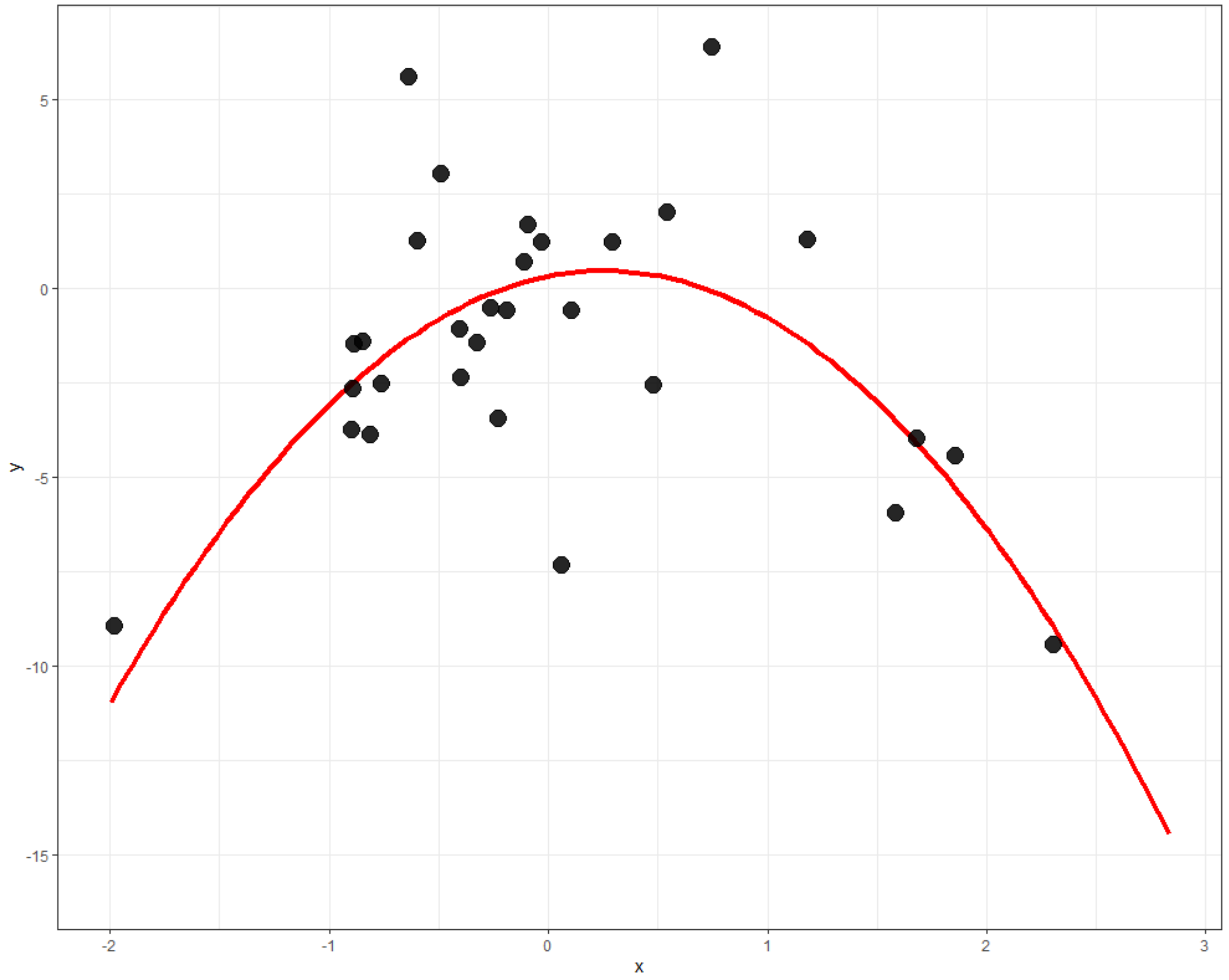
- For this specific problem, we collect $N = 30$ NOISY observations.
    - Denote the observed inputs as: $\mathbf{x} = \{x_1, x_2, \dots, x_n, \dots, x_N\}$
    - Denote observed responses as: $\mathbf{y} = \{y_1, y_2, \dots, y_n, \dots, y_N\}$
- Thus, we collect $N = 30$ input-output pairs:

$$\{x_n, y_n\}_{n=1}^{N=30}$$

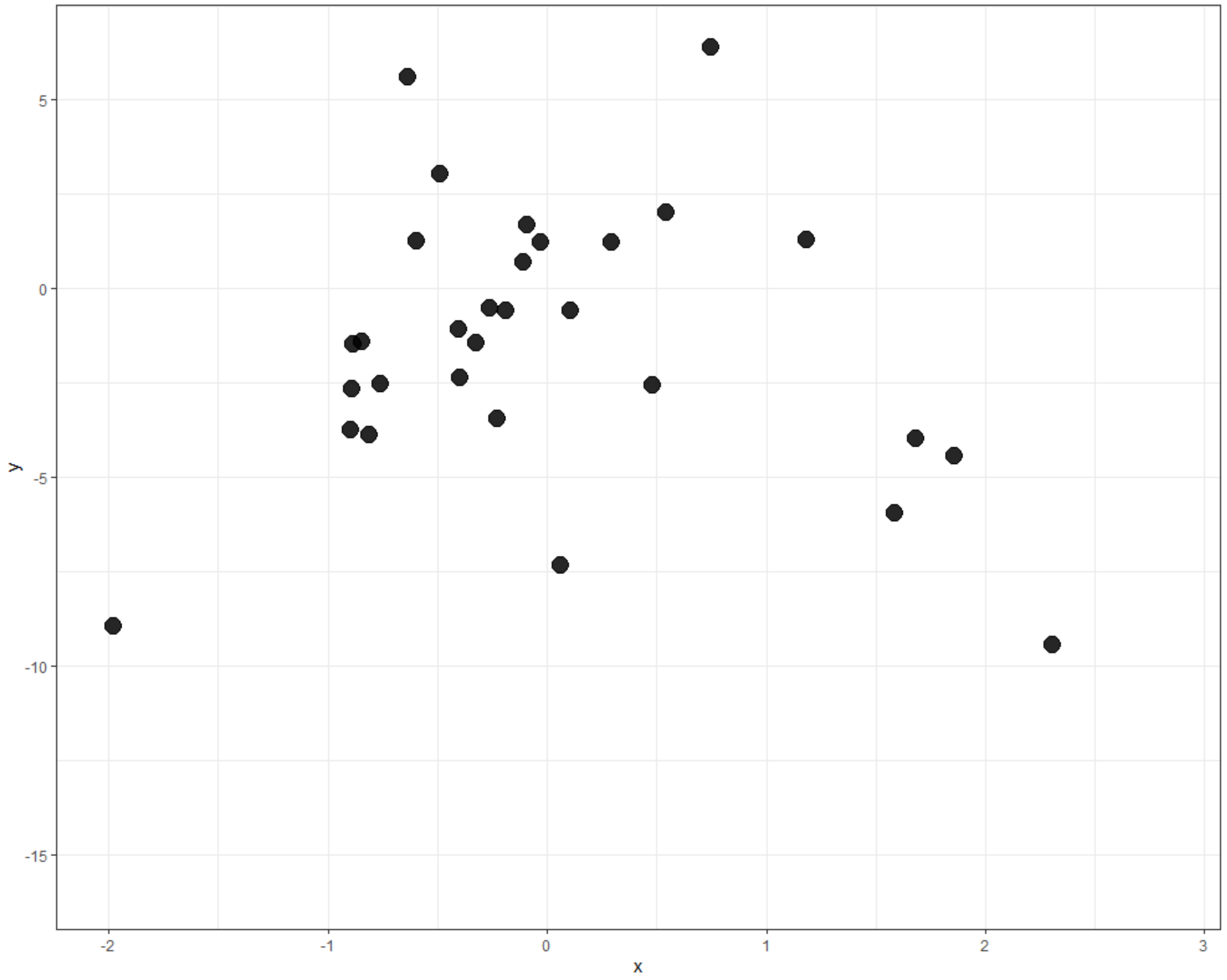
This is a toy problem. I am generating all observations with <u>**random number generators**</u>.

- We will discuss how that works during the **Distribution fitting** portion of the course.
- For now, just consider that 30 input-output pairs have been collected.
- We will find out HOW they were “collected” later.

Scatter plot showing the 30 noisy observations as black markers. TRUTH is still displayed as a red parabola.



If the TRUE trend wasn't shown…would you know the TRUTH is a parabola?



Let’s assume we do not know the TRUTH (just like a real problem)

- We want to **train** or ***fit*** a model between the response y and the input x.

- We will start out with a simple linear relationship:

$$y = \beta_0 + \beta_1 x + \text{error}$$

Let’s assume we do not know the TRUTH (just like a real problem)

- We want to **train** or ***fit*** a model between the response y and the input x.

- We will start out with a simple linear relationship:

$$y = \beta_0 + \beta_1 x + \boxed{\text{error}}$$

Remember, when learning from data we can only learn APPROXIMATE relationships!

You should expect there to always be ERROR!!!

Let’s assume we do not know the TRUTH (just like a real problem)

- We want to **train** or ***fit*** a model between the response y and the input x.

- We will start out with a simple linear relationship:

$$y = \beta_0 + \beta_1 x + \text{error}$$

The betas, β_0 and β_1, are the coefficients or parameters of our model.

We learn or ***estimate*** their values from the data by minimizing the ERROR.

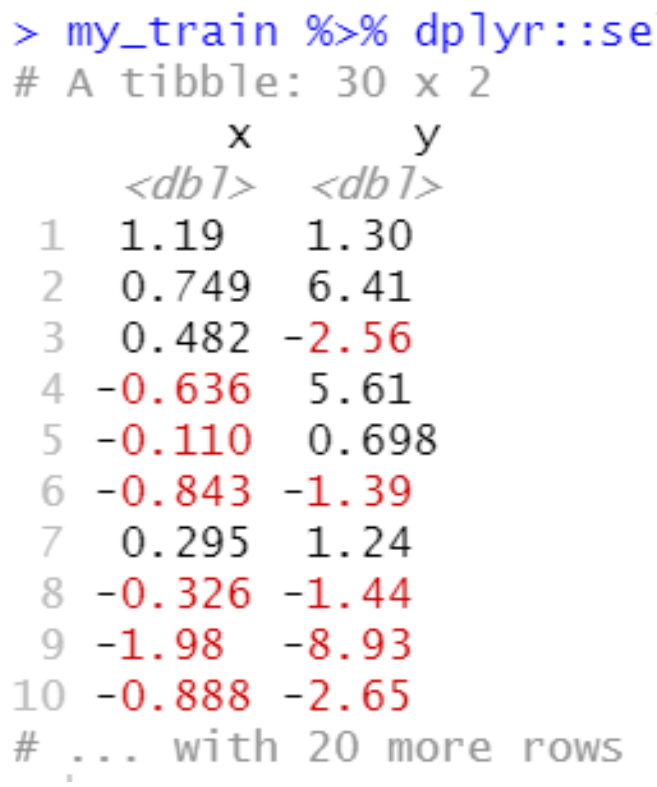
We will work through the exact details of the learning process later in the course.

- For now, let's just let `R` handle the heavy lifting for us.

- Store the data in the object `my_train` which contains two columns (variables) named `x` and `y`.

- Fit a linear relationship in `R` using the `lm()` function and the formula interface.

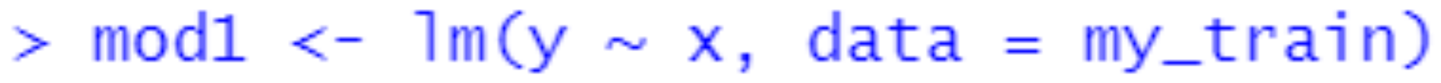
Print out a few rows of the `my_train` object to show the variables we will work with


```
> my_train %>% dplyr::select(x, y)
# A tibble: 30 x 2
        x      y
    <dbl>  <dbl>
 1  1.19   1.30
 2  0.749  6.41
 3  0.482 -2.56
 4 -0.636  5.61
 5 -0.110  0.698
 6 -0.843 -1.39
 7  0.295  1.24
 8 -0.326 -1.44
 9 -1.98  -8.93
10 -0.888 -2.65
# ... with 20 more rows
```

- Please see the *Introduction to R* videos available on Canvas to learn about the pipe operator, `%>%`, and the `tidyverse` package `dplyr` for manipulating and modifying `data.frames` and `tibbles`.

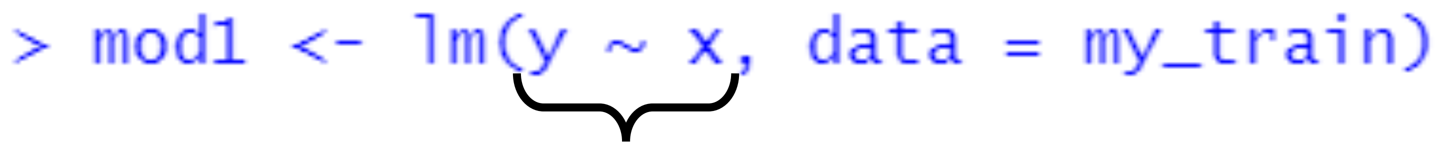
- The R4DS book provides an excellent introduction to `dplyr`.

Fit the model with `lm()`



```
> mod1 <- lm(y ~ x, data = my_train)
```

Fit the model with `lm()`



```
> mod1 <- lm(y ~ x, data = my_train)
```

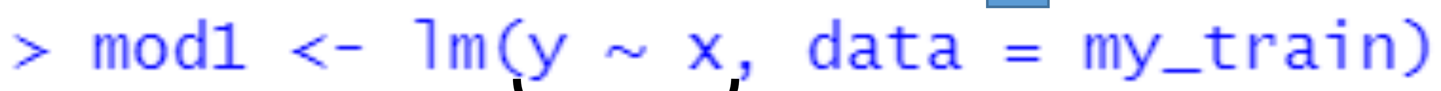
Formula interface allows you to specify the response and inputs (more formally the predictors) to the model.

Basic expression: `<output variable names> ~ <input variable name>`

Read the expression as: “the output, `y`, is a function of the input, `x`”

Fit the model with `lm()`

Assign the `my_train` object to the `data` argument to the `lm()` function



```
> mod1 <- lm(y ~ x, data = my_train)
```

Formula interface allows you to specify the response and inputs (more formally the predictors) to the model.

Basic expression: `<output variable names> ~ <input variable name>`

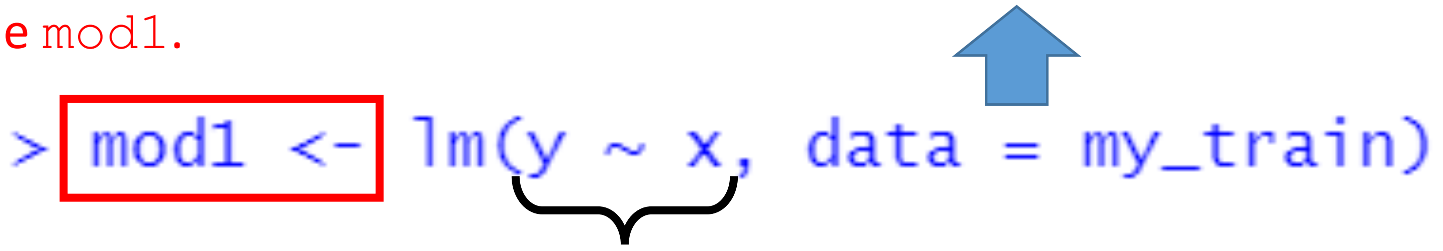
Read the expression as: “the output, `y`, is a function of the input, `x`”

Fit the model with `lm()`

The assignment operator, `<-`, assigns an object to a variable.

The result of the `lm()` call is assigned to the variable `mod1`.

Assign the `my_train` object to the `data` argument to the `lm()` function



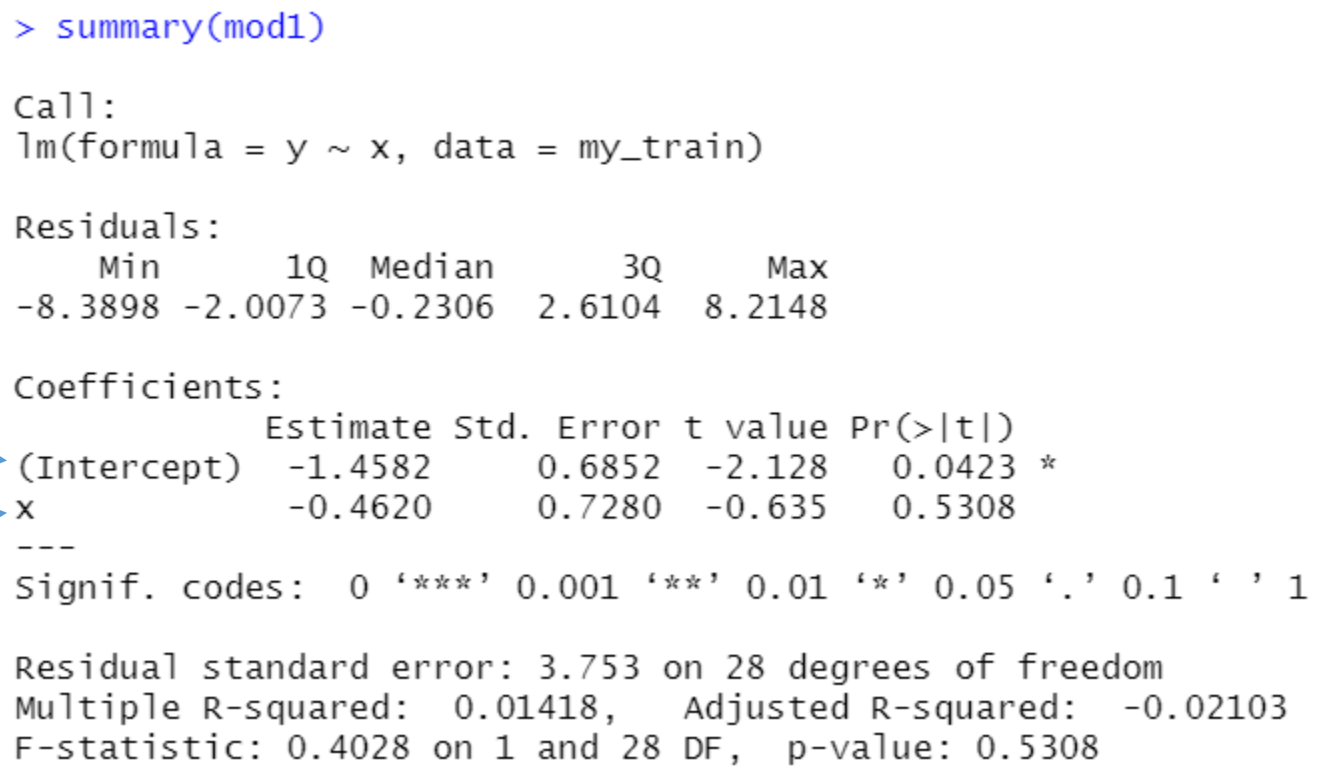
```
> mod1 <- lm(y ~ x, data = my_train)
```

Formula interface allows you to specify the response and inputs (more formally the predictors) to the model.

Basic expression: `<output variable names> ~ <input variable name>`

Read the expression as: "the output, `y`, is a function of the input, `x`"

We can summarize the results with the `summary()` function



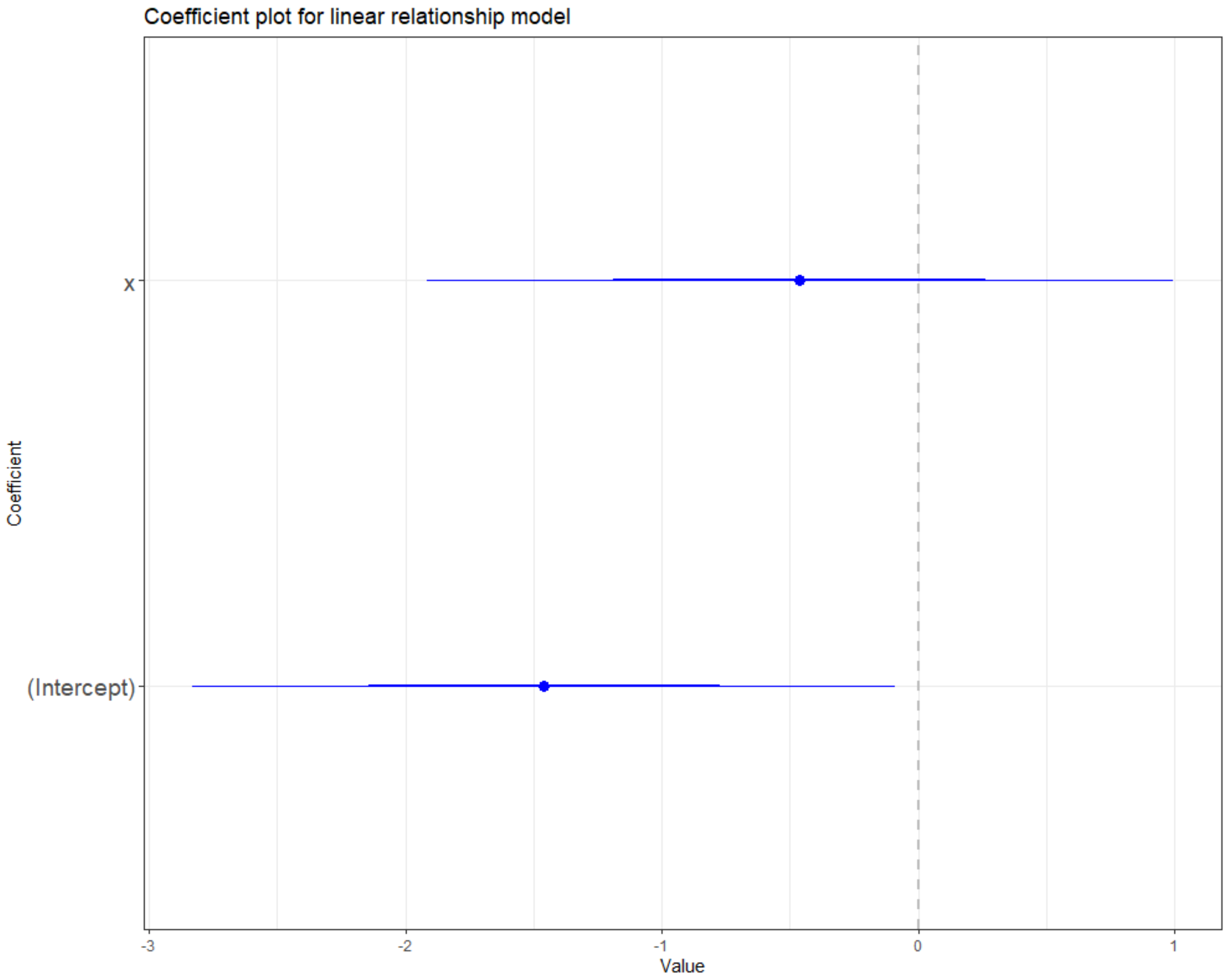
```
> summary(mod1)

Call:
lm(formula = y ~ x, data = my_train)

Residuals:
    Min      1Q  Median      3Q     Max 
-8.3898 -2.0073 -0.2306  2.6104  8.2148 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)  
(Intercept)  -1.4582     0.6852  -2.128   0.0423 *
x            -0.4620     0.7280  -0.635   0.5308  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.753 on 28 degrees of freedom
Multiple R-squared:  0.01418,	Adjusted R-squared:  -0.02103 
F-statistic: 0.4028 on 1 and 28 DF,  p-value: 0.5308
```

β_0

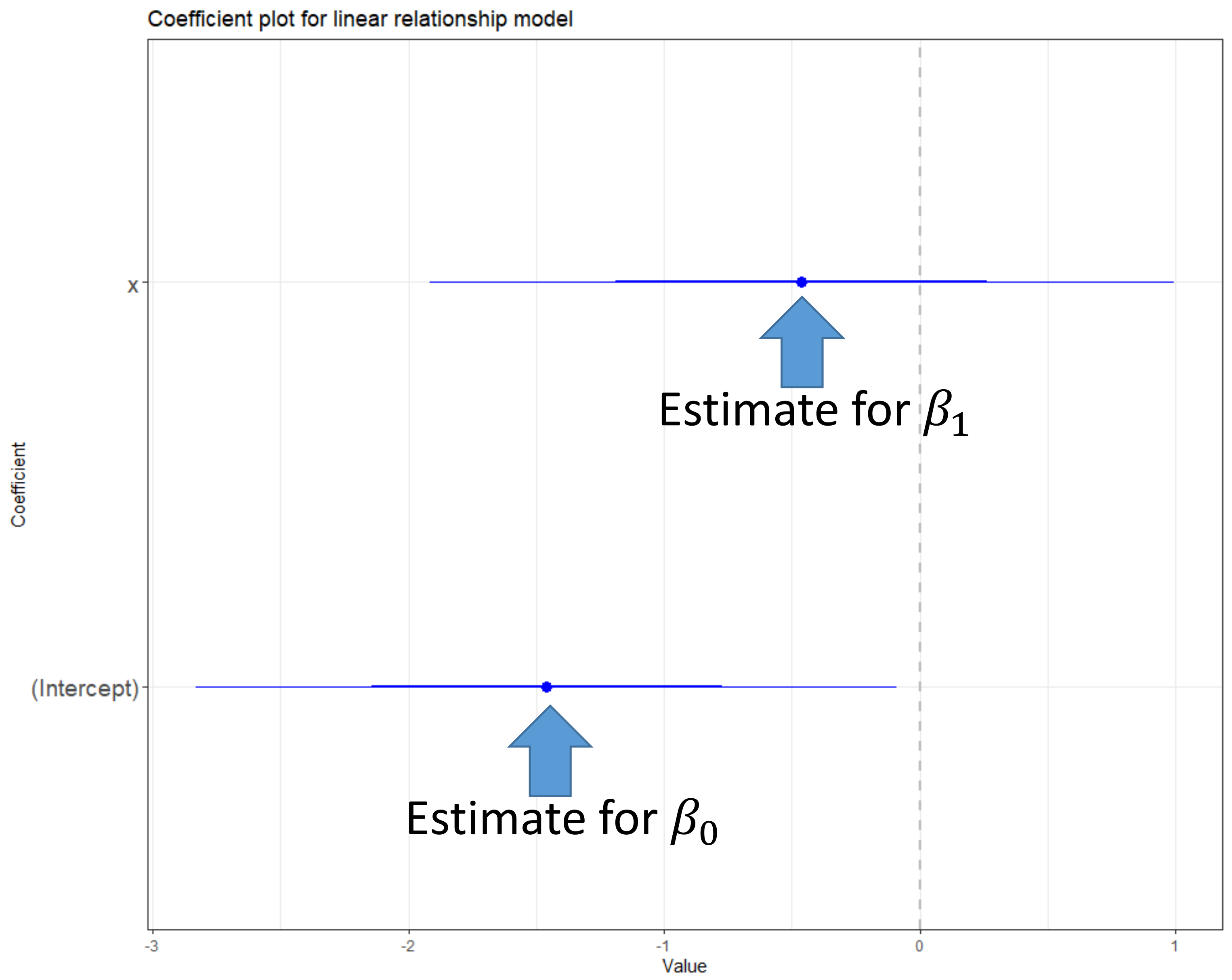
β_1

You do NOT need to know what these mean right now.

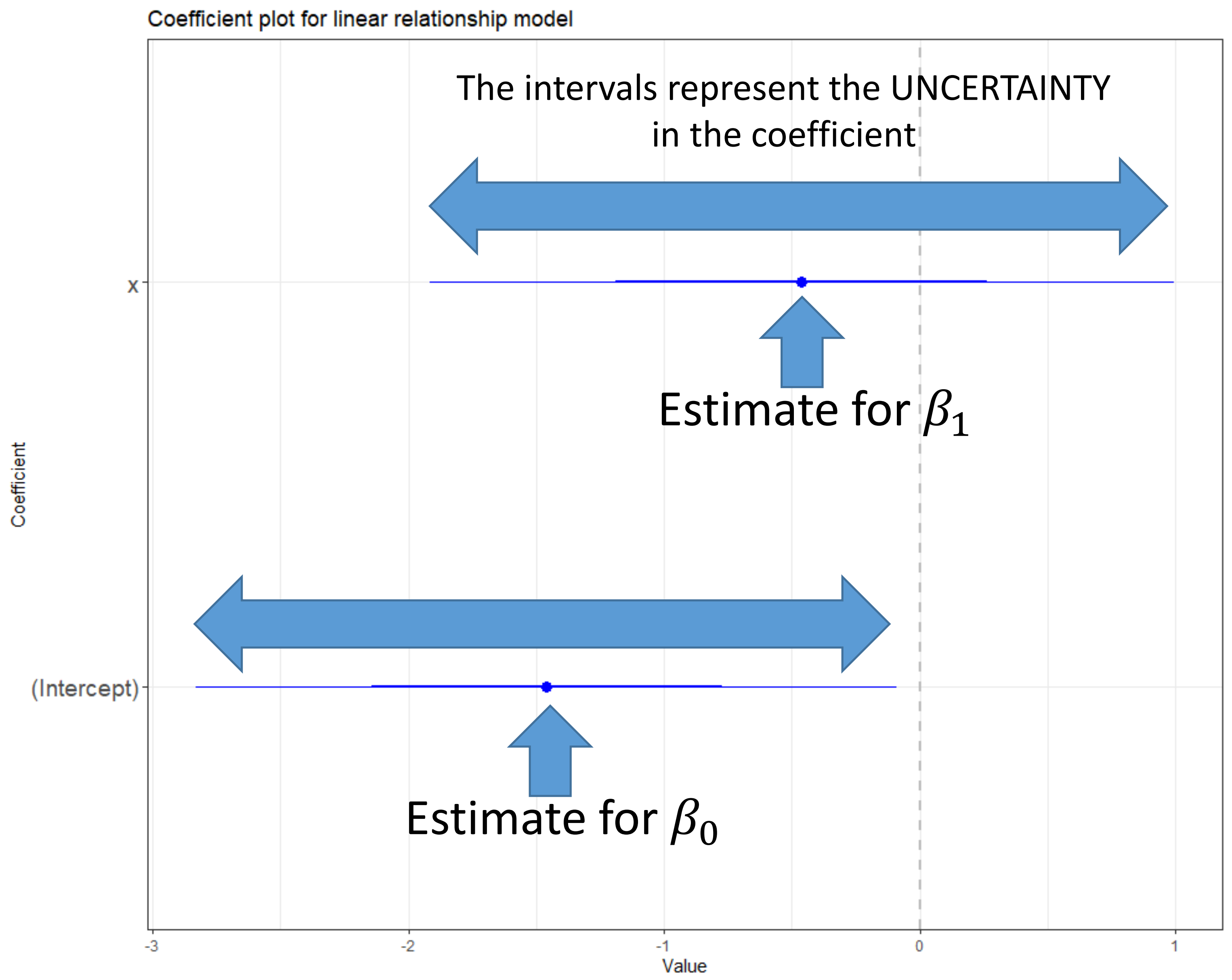
I prefer to visualize the coefficient estimates and confidence intervals.



I prefer to visualize the coefficient estimates and confidence intervals.



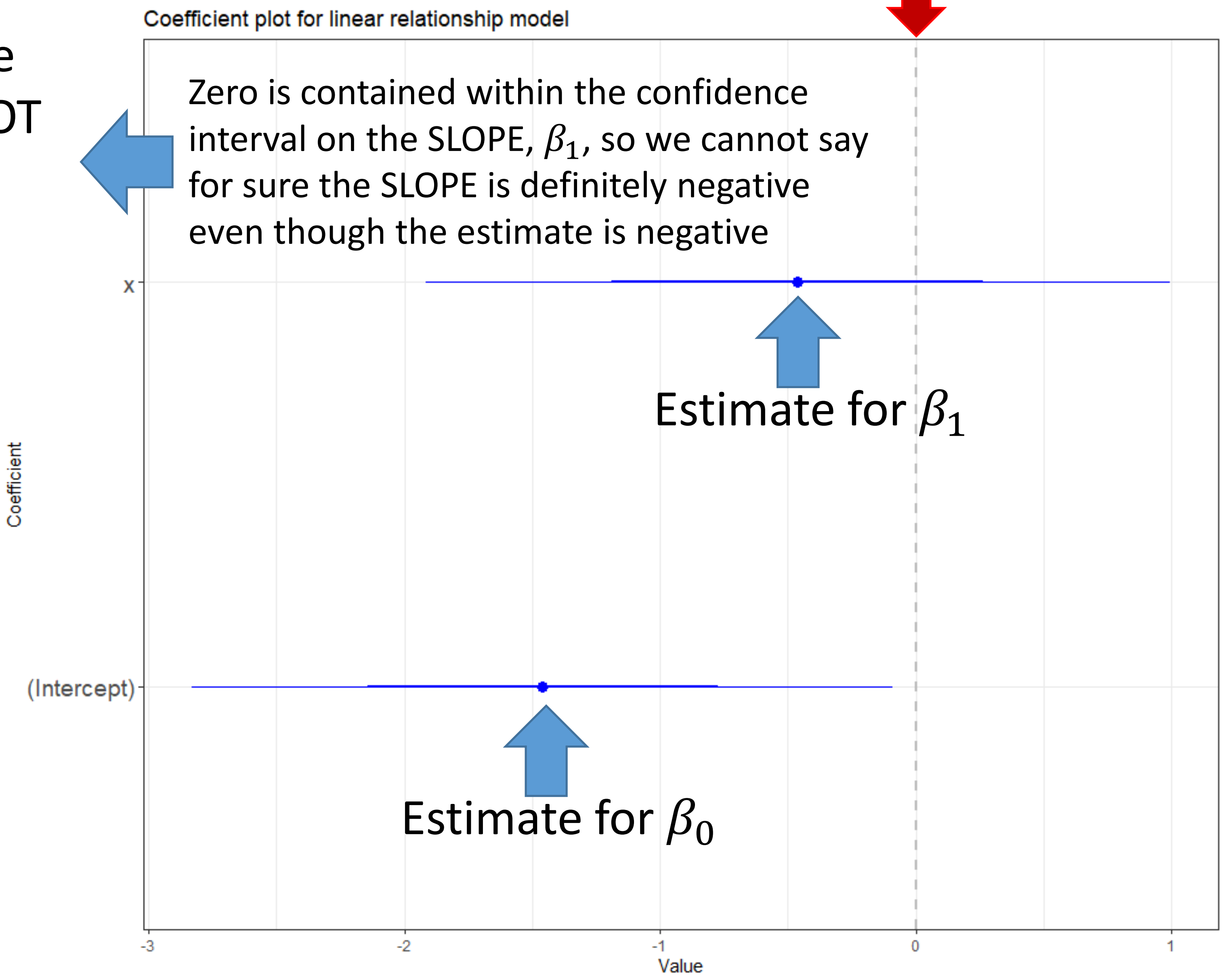
I prefer to visualize the coefficient estimates and confidence intervals.



I prefer to visualize the coefficient es ꞏꞏꞏ e intervals.

Coefficient =0

x is therefore considered NOT **statistically significant**



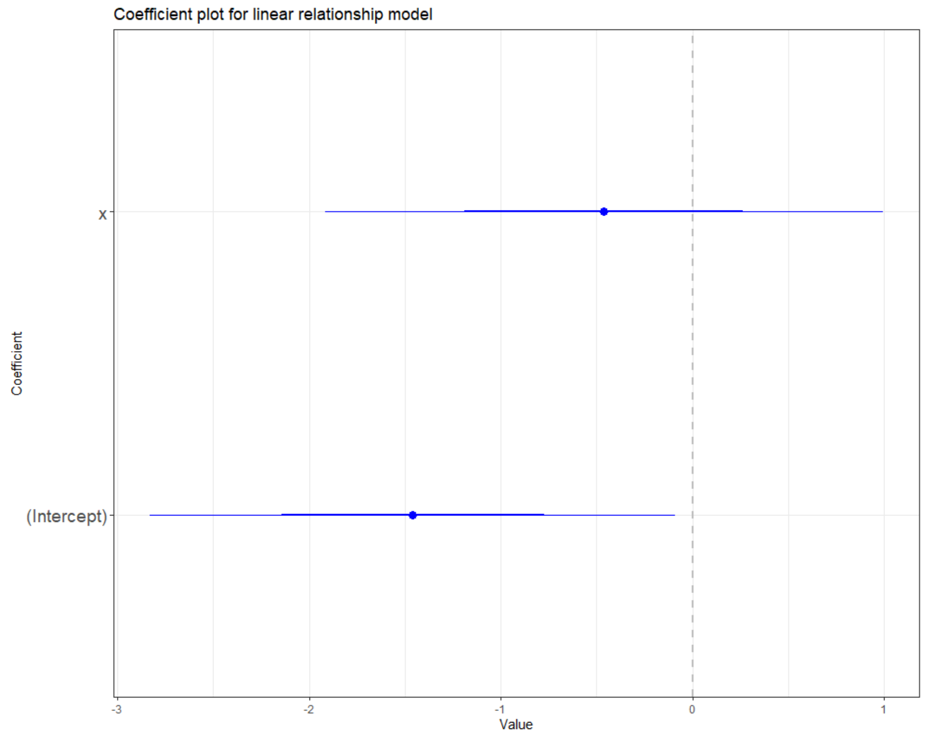
Zero is contained within the confidence interval on the SLOPE, β_1, so we cannot say for sure the SLOPE is definitely negative even though the estimate is negative

Estimate for β_1

Estimate for β_0

This figure is created by passing the `mod1` object into the `coefplot()` function from the `coefplot` package.

```
coefplot::coefplot(mod1) +
  labs(title = "Coefficient plot for linear relationship model") +
  theme_bw() +
  theme(axis.text.y = element_text(size = 14))
```



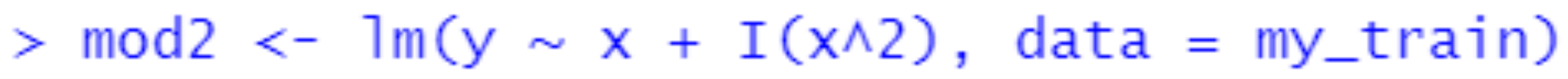
But…that’s just one model!

- What if we ***fit*** a **quadratic relationship** between the response and the input?

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \text{error}$$

- All we need to do is change what we type into the formula interface.

Fit a quadratic relationship with `lm()`



```
> mod2 <- lm(y ~ x + I(x^2), data = my_train)
```

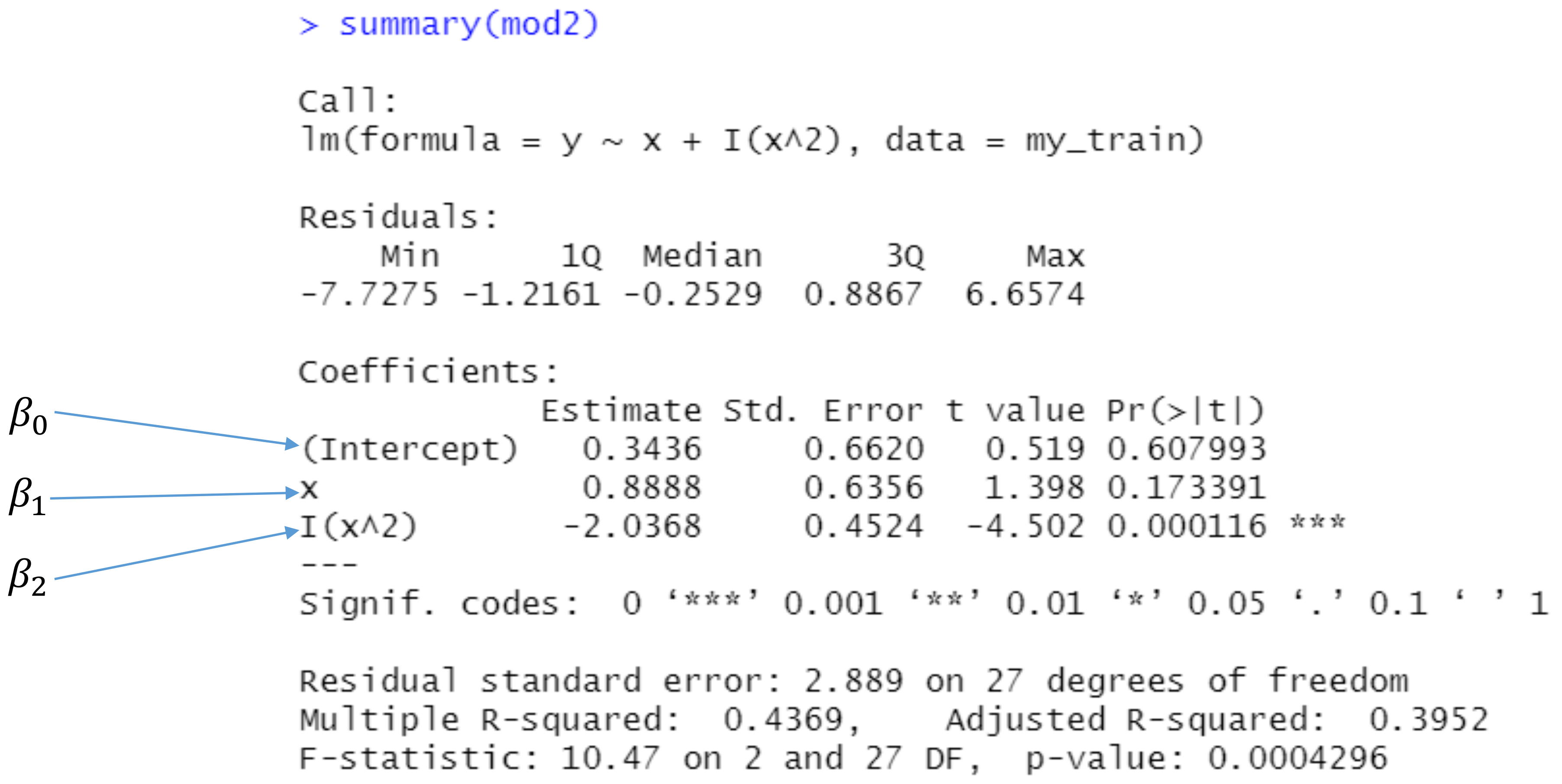
The formula interface is "read" the same as before, but now our model has more PREDICTORS or FEATURES!

We still have **1 input**, x, but **TWO PREDICTORS**: x and x^2.

ADDITIVE formula expression with 2 predictors: `<output> ~ <predictor 1> + <predictor 2>`

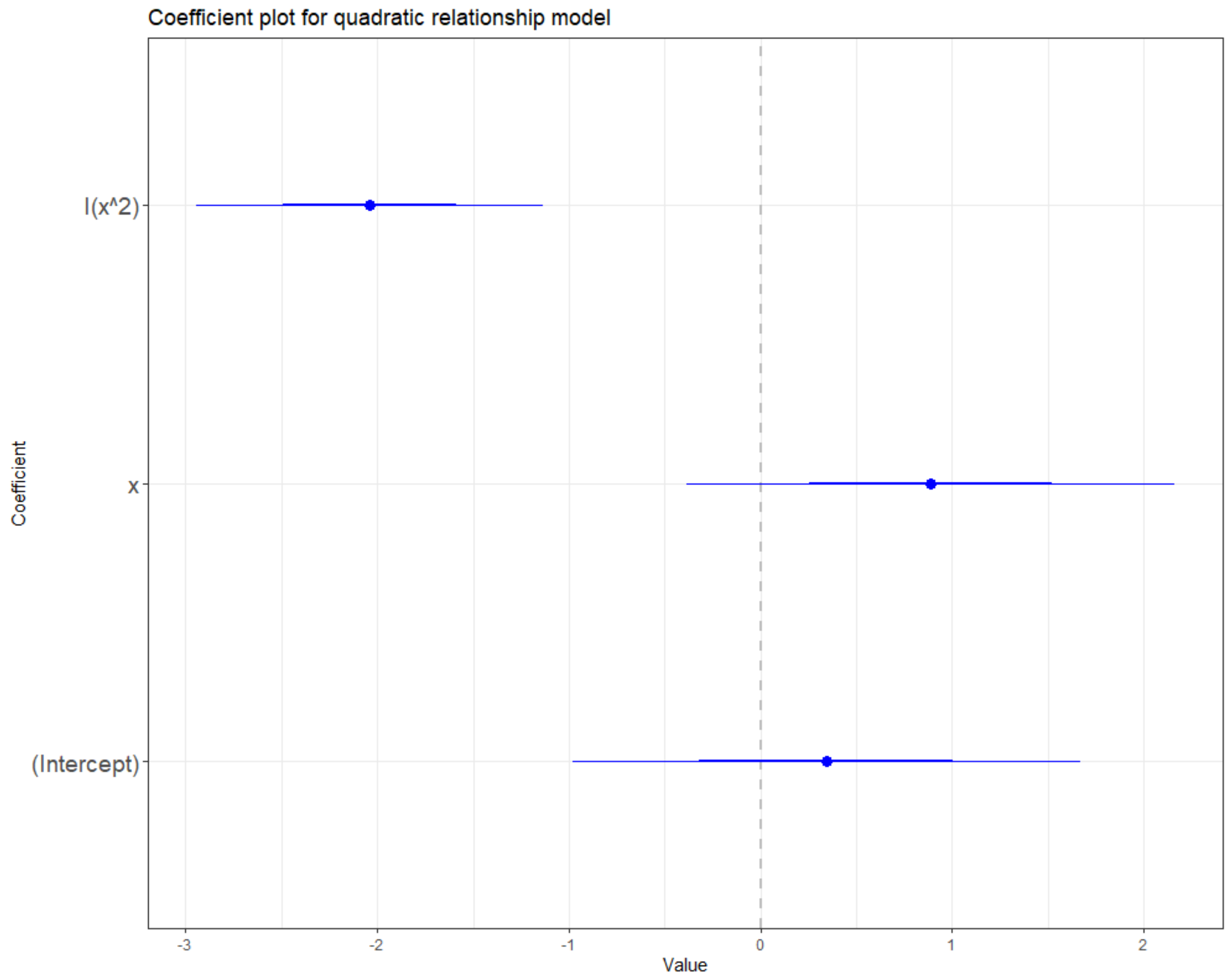
In `R`, we must place the x^2 predictor within the `I()` function. We will learn that `^2` is a special shortcut…so the `I()` function tells the formula interface to use the expression "as is".

We can summarize the quadratic relationship with `summary()` just as we did before



```
> summary(mod2)

Call:
lm(formula = y ~ x + I(x^2), data = my_train)

Residuals:
    Min      1Q  Median      3Q     Max 
-7.7275 -1.2161 -0.2529  0.8867  6.6574 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   0.3436     0.6620   0.519 0.607993    
x             0.8888     0.6356   1.398 0.173391    
I(x^2)       -2.0368     0.4524  -4.502 0.000116 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.889 on 27 degrees of freedom
Multiple R-squared:  0.4369,	Adjusted R-squared:  0.3952 
F-statistic: 10.47 on 2 and 27 DF,  p-value: 0.0004296
```

As before, let’s visualize the coefficients



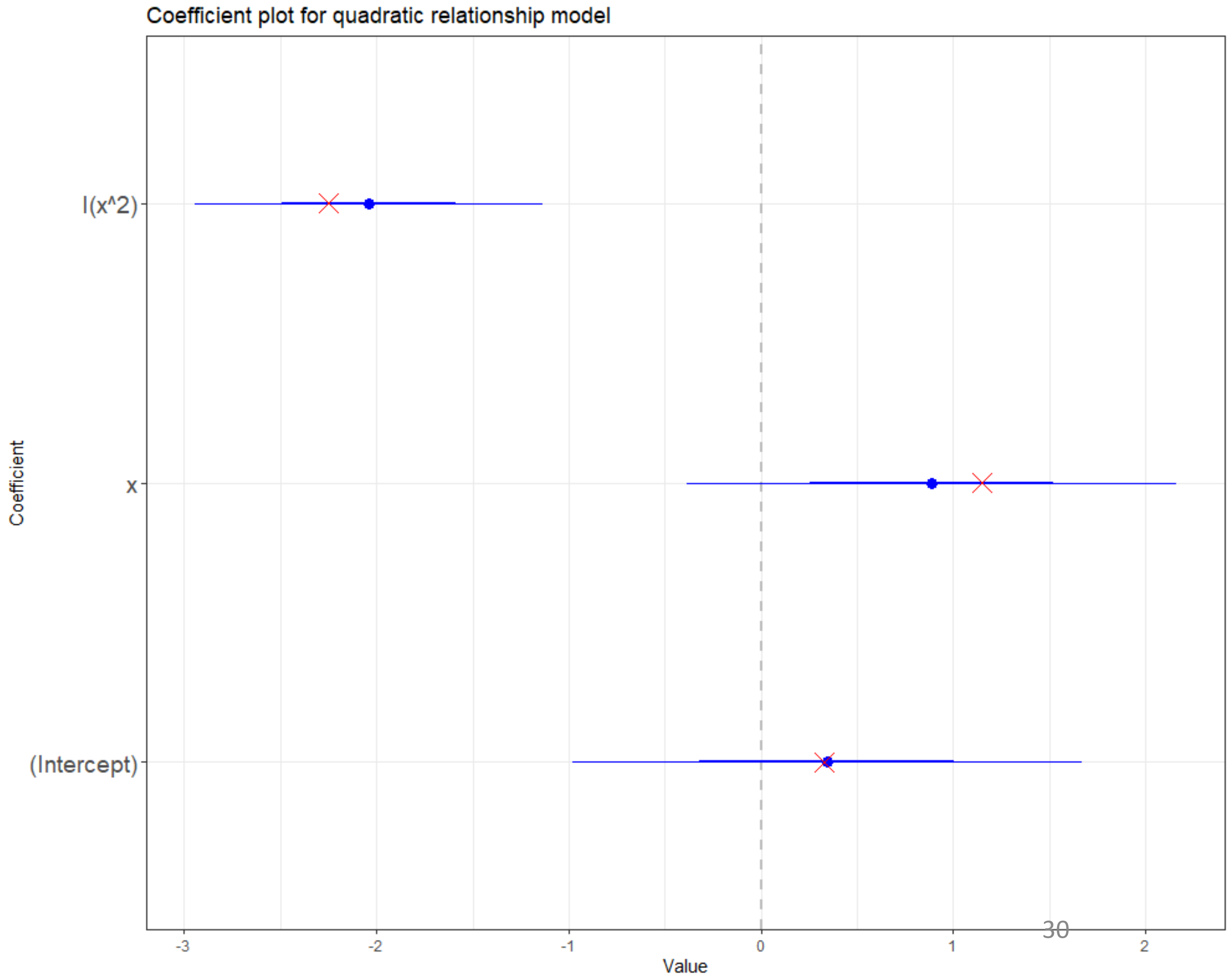
Remember this is a toy problem, we know the TRUE coefficient values associated with the TRUE trend!

We can compare the estimates to their TRUE values for this example.

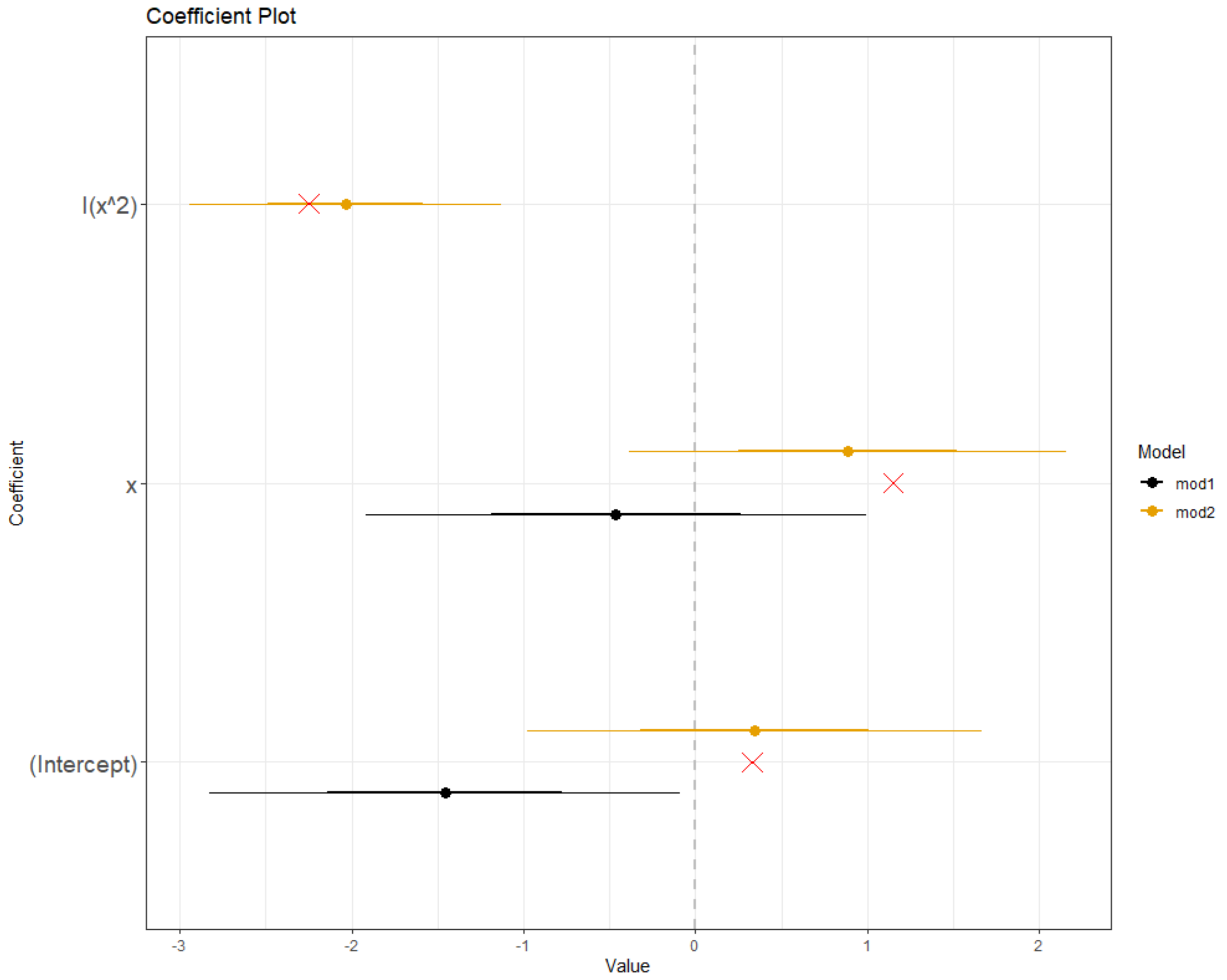
$$\beta_{*0} = 0.33, \beta_{*1} = 1.15, \text{and } \beta_{*2} = -2.25$$

The TRUE coefficient values are displayed as red x's in the figure to the right.

Not only are the estimates close to the TRUE values, but the TRUE values are "contained" within the confidence intervals!



Compare the coefficients between the two models



Which model is better?

Which model is better?

- Since we cannot compare the coefficient estimates to TRUE values in a real problem, how can we assess which model is better?

Which model is better?

- Since we cannot compare the coefficient estimates to TRUE values in a real problem, how can we assess which model is better?
- **We need a performance metric**!
- Remember we stated that the coefficients are estimated by minimizing the error.
- Specifically, the **sum of squared errors** between the model and the observations (this is where the phrase Least Squares comes from!)

Regression performance metrics

- A natural choice for the performance metric in regression problems is the **Mean Squared Error** (MSE).
  - The mean or average squared error across all observations.

- The MSE is not in the same units as the response,
  - Take the square root of the MSE to put the performance metric in the same units as the response
  - So, it is common to consider the square **Root Mean Squared Error** (RMSE) as a performance metric.

- Alternatively, we could also consider the **Mean Absolute Error** (MAE).

But why stop at just 2 models?

- Why can’t we try a higher degree polynomial?

- For example, what if we tried a cubic or a 5th degree polynomial?

But why stop at just 2 models?

- Why can’t we try a higher degree polynomial?

- For example, what if we tried a cubic or a 5th degree polynomial?

- **Let’s compare a total of 9 models**.

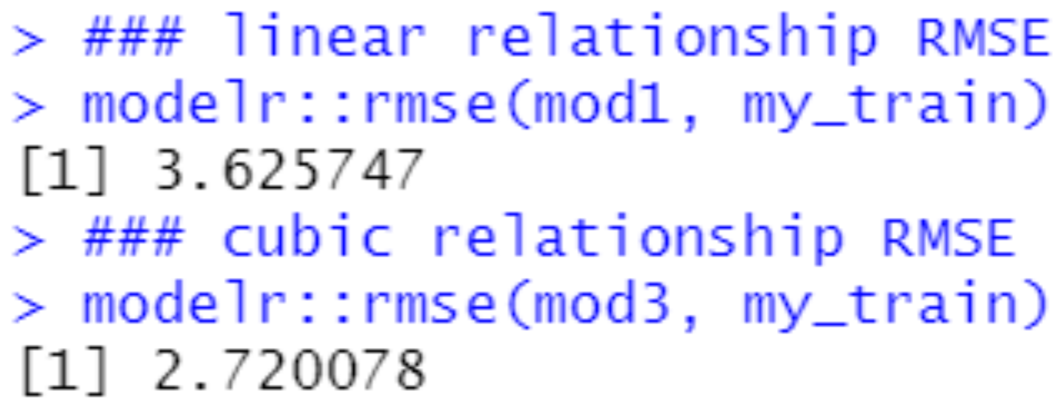
- 0th degree (intercept-only or constant) model up to an 8th degree polynomial!

Can simply type the appropriate formula for each of the desired models

```r
### this is not the efficient way to do this...

mod0 <- lm(y ~ 1, data = my_train)

mod3 <- lm(y ~ x + I(x^2) + I(x^3), data = my_train)

mod4 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4), data = my_train)

mod5 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5), data = my_train)

mod6 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6), data = my_train)

mod7 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7), data = my_train)

mod8 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7) + I(x^8), data = my_train)
```

Calculate the RMSE associated with each model

- Can use the `rmse()` function from the `modelr` package
  - `modelr` comes with the `tidyverse` so if you installed the `tidyverse` you already have `modler`!
- Syntax: `rmse(<model object>, <data to calculate error relative to>)`
- For example, to calculate the RMSE for the linear relationship and cubic relationship models:



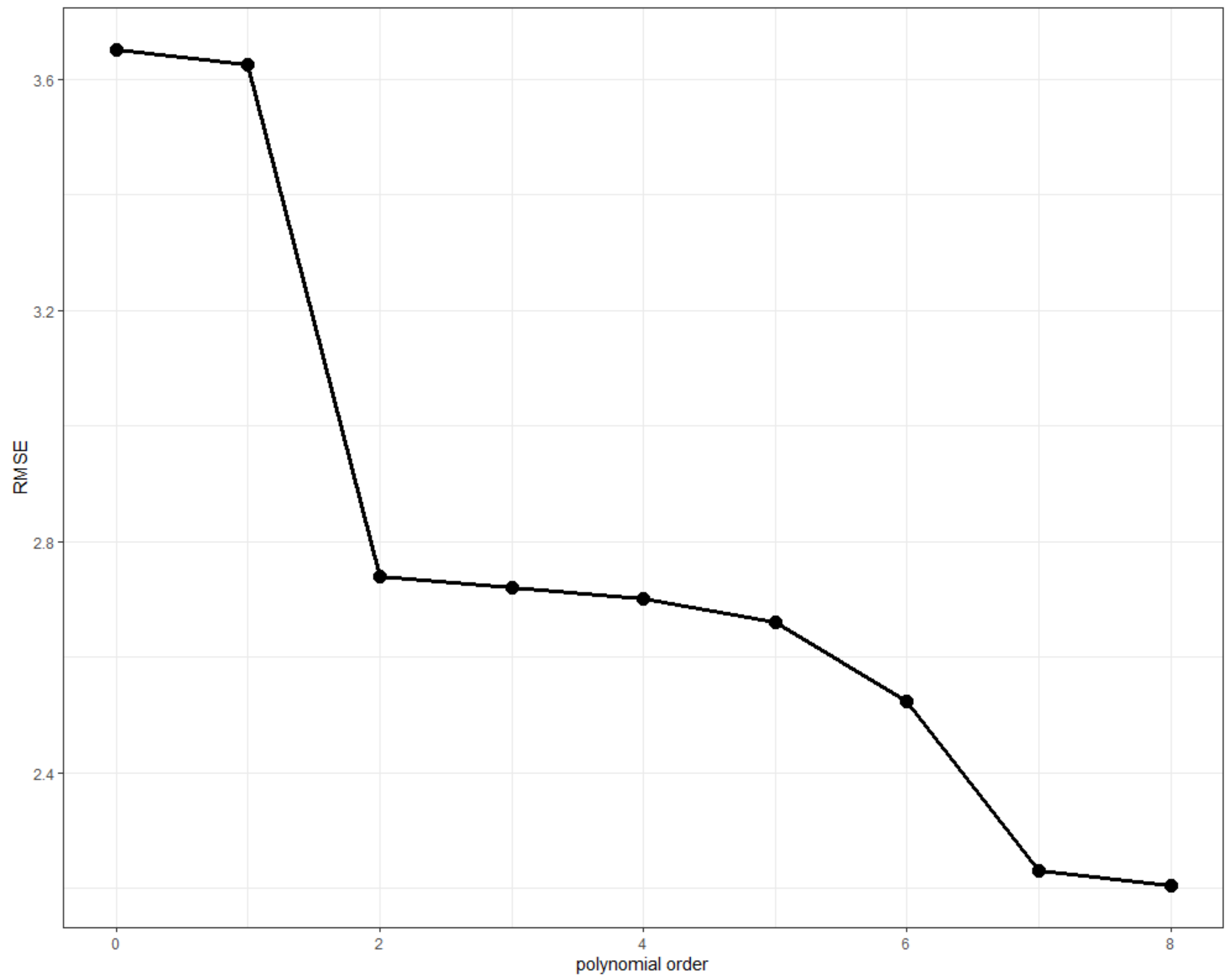
```
> ### linear relationship RMSE
> modelr::rmse(mod1, my_train)
[1] 3.625747
> ### cubic relationship RMSE
> modelr::rmse(mod3, my_train)
[1] 2.720078
```

Calculate the RMSE for all 9 models

- Calculate the RMSE relative to the training set, `my_train`.

- Which model do you think will be the best?

The 8th degree polynomial has the lowest RMSE!!



What’s going on…after all we know the TRUE trend is a parabola!

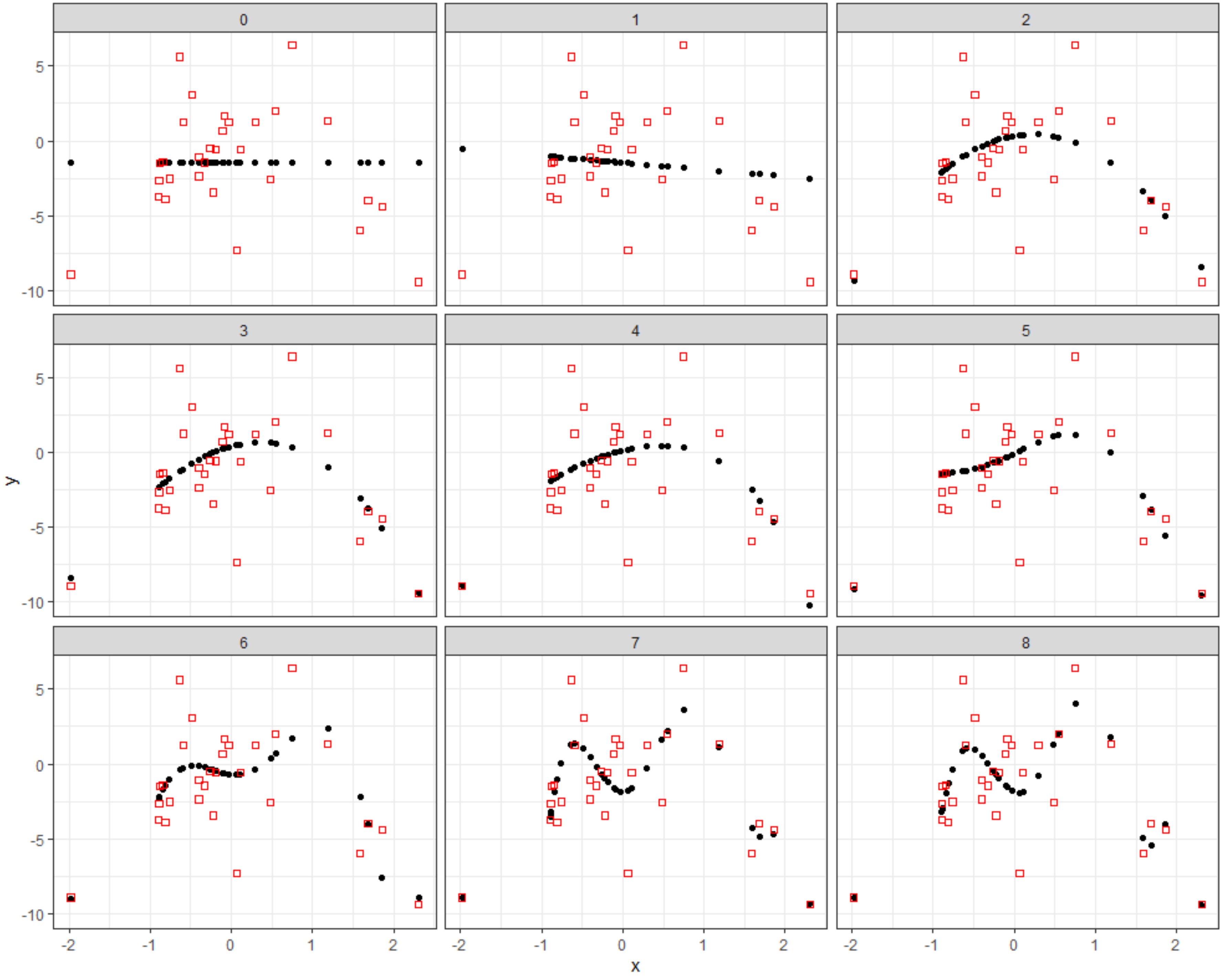
What's going on…after all we know the TRUE trend is a parabola!

- Let's look at the model "fits" or predictions on the training set.

- The next figure shows a separate subplot or ***facet*** for each model.
  - The facet strip displays the polynomial degree

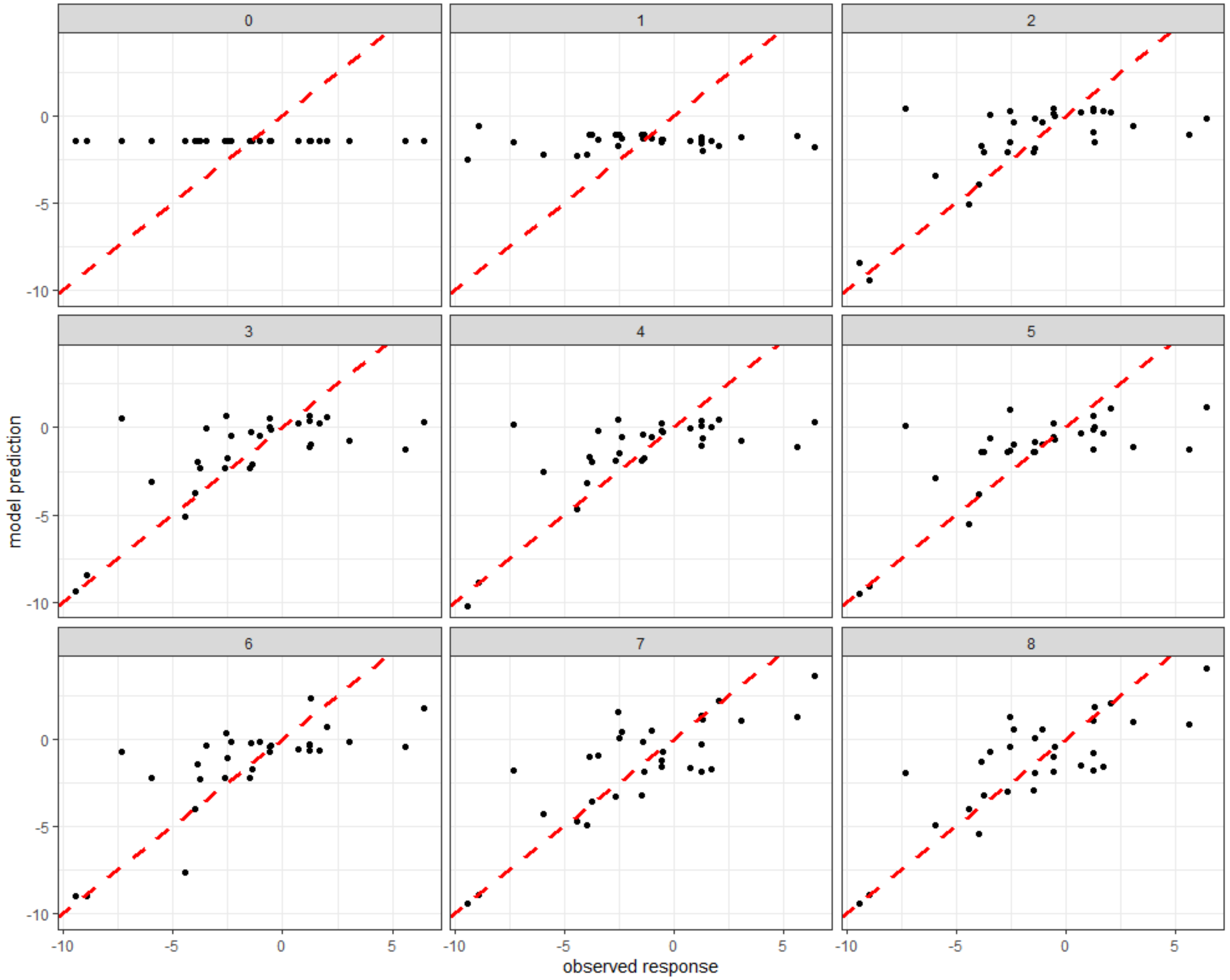
- The training output are shown as red open squares and the model "**fits**" are displayed as **black dots**.

- The response is visualized with respect to the input.

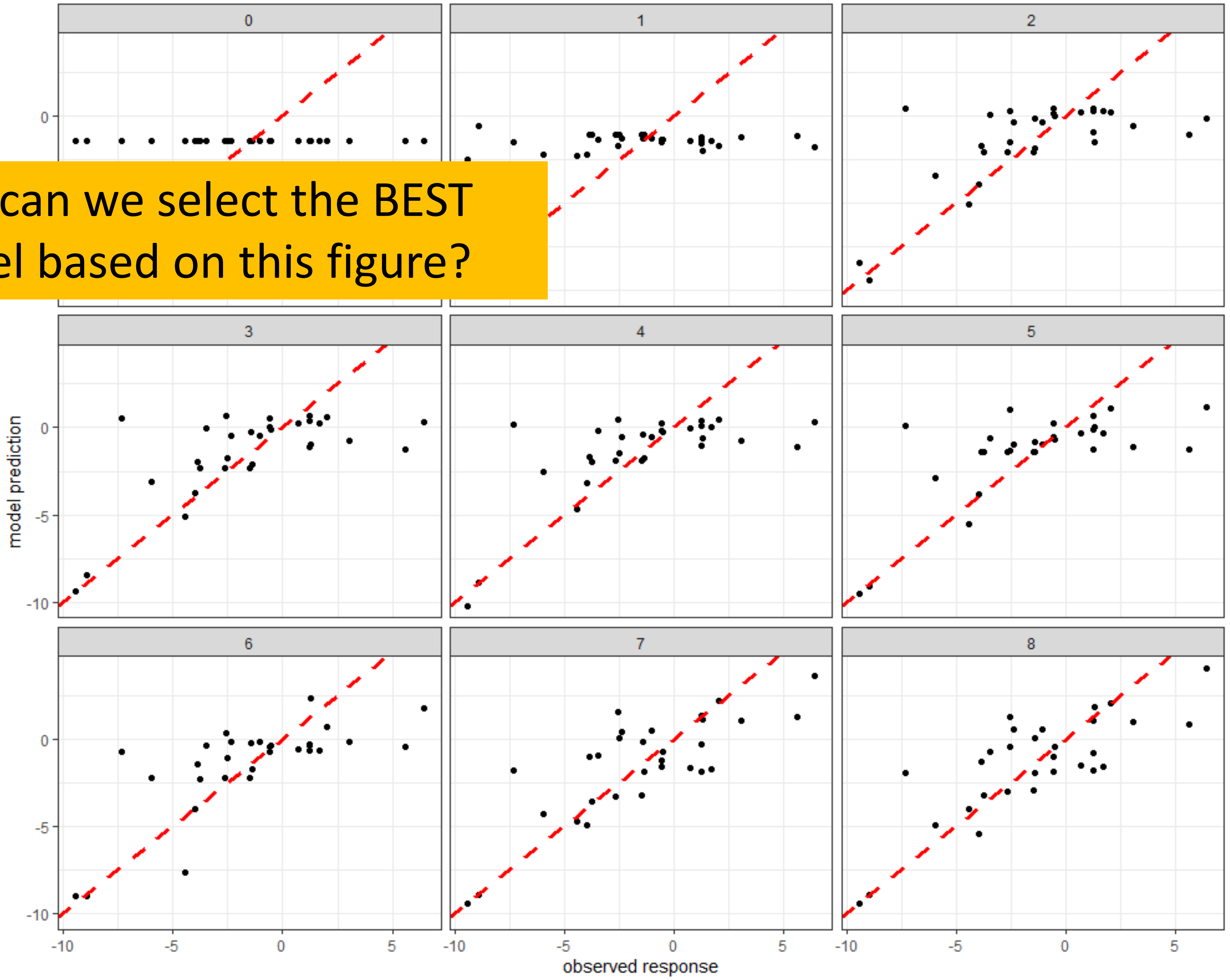

0
1
2
3
4
5
6
7
8
y
x
5
0
-5
-10
-2
-1
0
1
2

Rather than visualizing the “fits” vs the input…

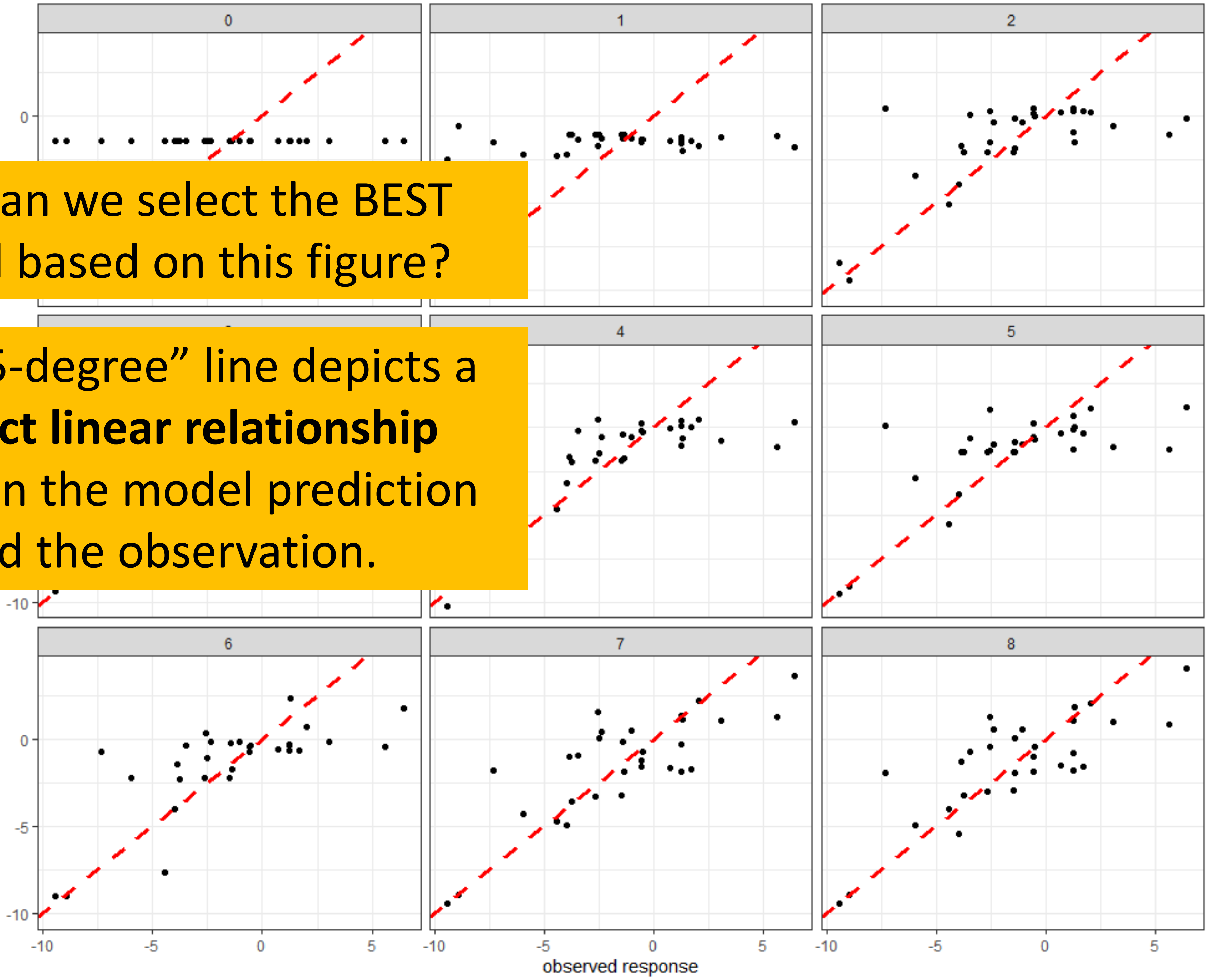
- Let’s visualize the “fits” or model training set predictions with respect to the observed training output.
- The following figure is known as the **predicted vs observed figure**.
- As with the previous figure, each facet is a separate polynomial degree.



0
1
2
3
4
5
6
7
8
model prediction
observed response
-10
-5
0
5

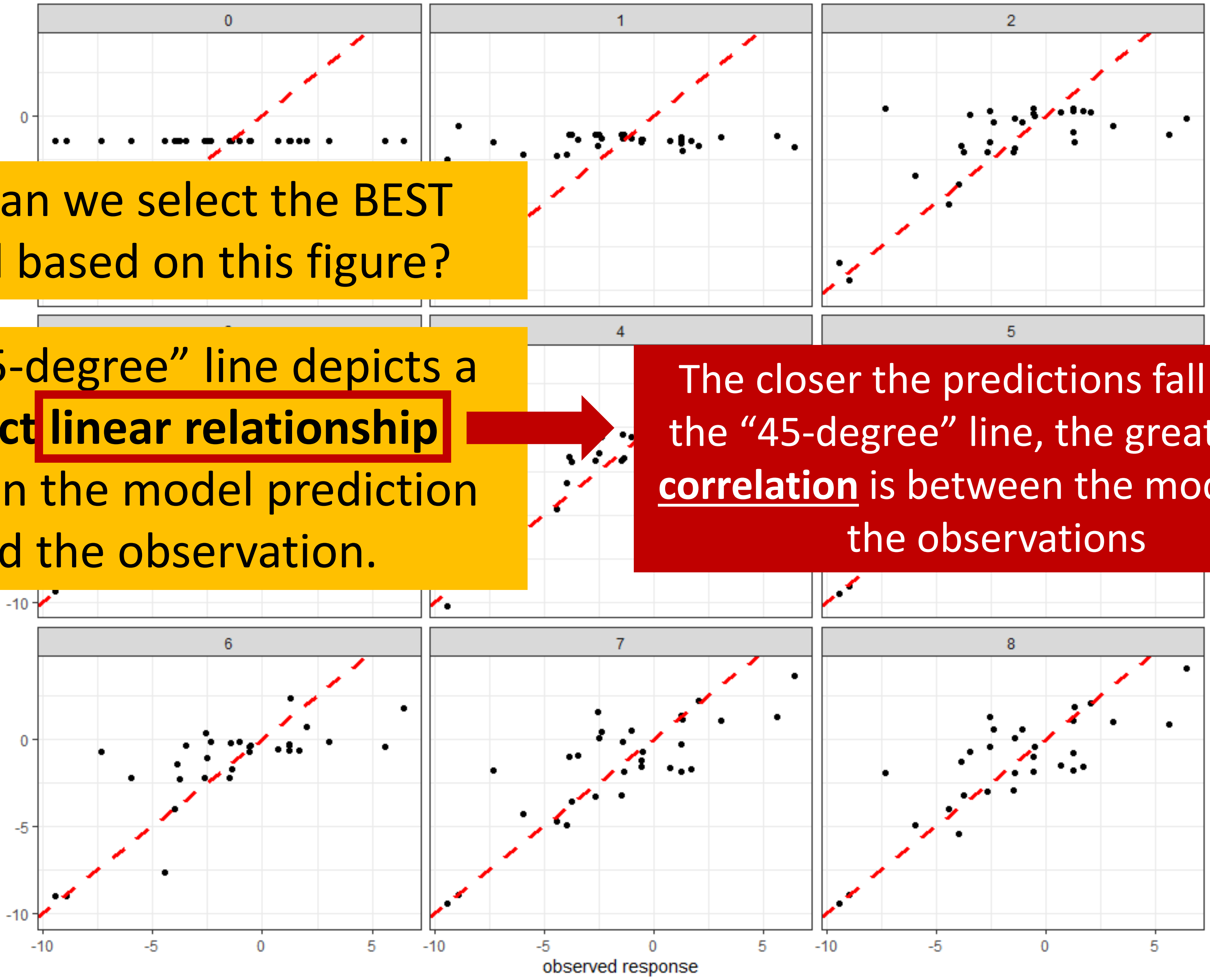


How can we select the BEST model based on this figure?



How can we select the BEST model based on this figure?

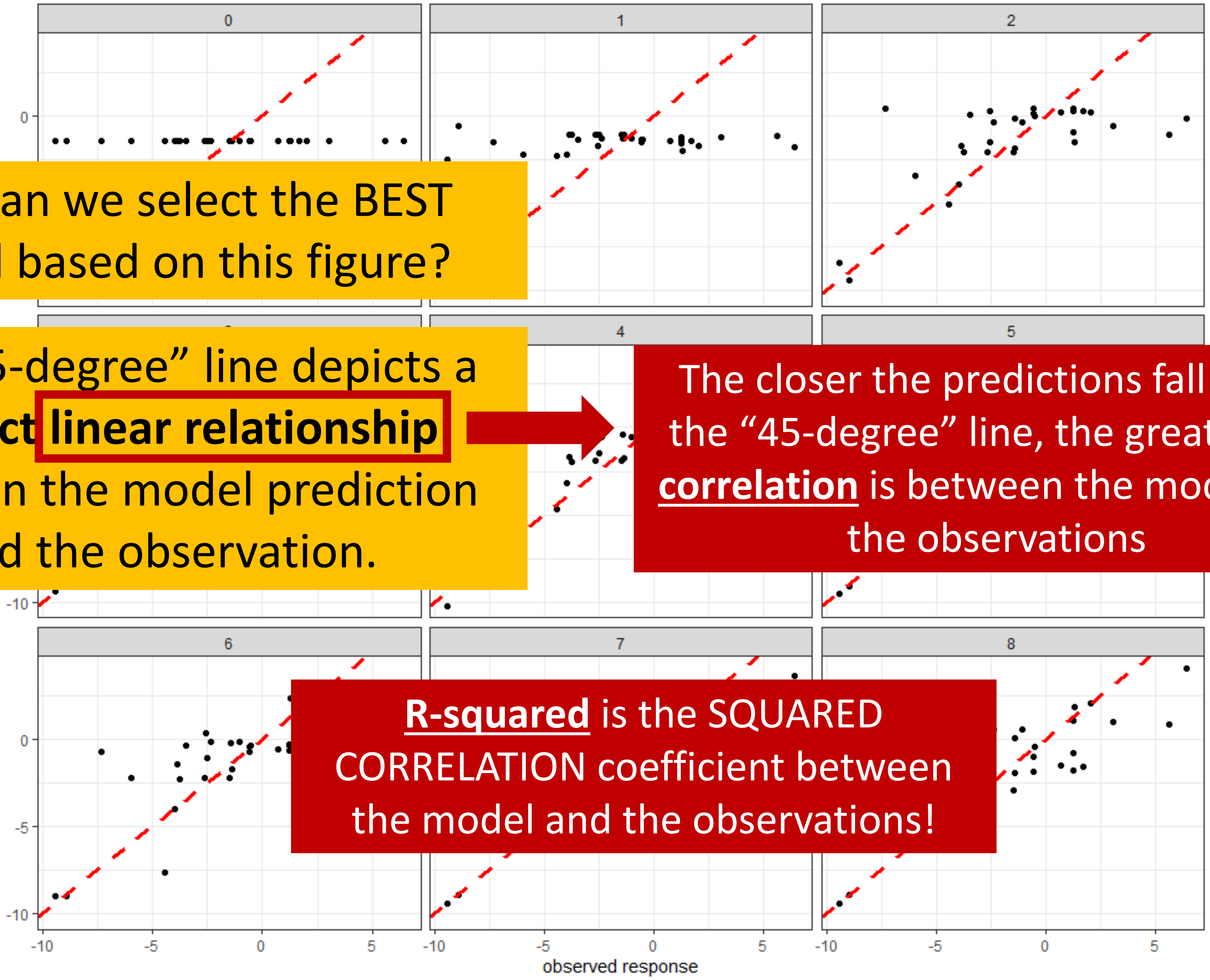
The "45-degree" line depicts a **perfect linear relationship** between the model prediction and the observation.



How can we select the BEST model based on this figure?

The "45-degree" line depicts a **perfect linear relationship** between the model prediction and the observation.

The closer the predictions fall along the "45-degree" line, the greater the **correlation** is between the model and the observations



How can we select the BEST model based on this figure?

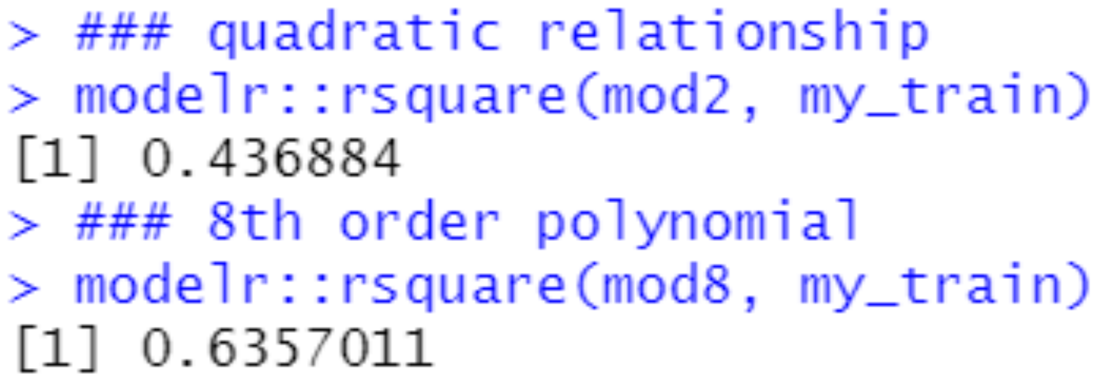
The "45-degree" line depicts a **perfect linear relationship** between the model prediction and the observation.

The closer the predictions fall along the "45-degree" line, the greater the **correlation** is between the model and the observations

R-squared is the SQUARED CORRELATION coefficient between the model and the observations!

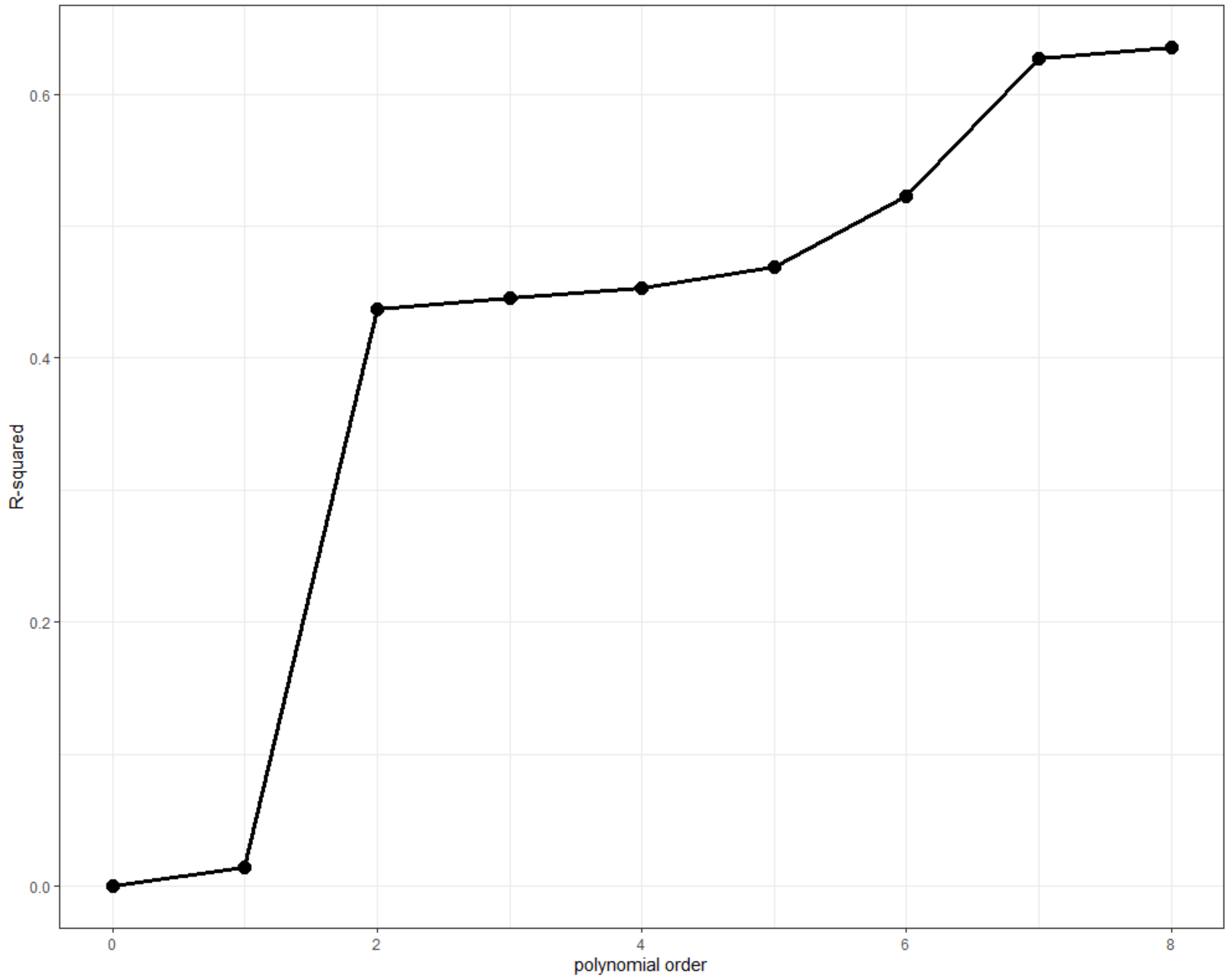
Calculate R-squared using the `modelr` function `rsquare()` with the same syntax as `rmse()`

- For example, to calculate R-squared associated with the quadratic relationship and the 8th degree polynomial relative to the training set:



```
> ### quadratic relationship
> modelr::rsquare(mod2, my_train)
[1] 0.436884
> ### 8th order polynomial
> modelr::rsquare(mod8, my_train)
[1] 0.6357011
```

Calculate R-squared for all 9 models…the 8th degree model is the best!



Performance metrics are improving as the polynomial degree increases!

- The polynomial degree represents COMPLEXITY.
- Higher polynomial degree means the model has **MORE** coefficients.
- **More coefficients means greater complexity!!!!**
- **The performance is getting better as the models become more and more complex!**

We have assessed model performance with respect to the TRAINING SET

- How well do the models generalize to NEW data?

We have assessed model performance with respect to the TRAINING SET

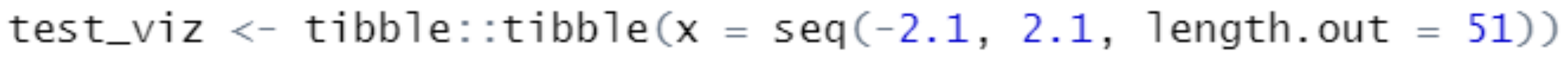
- How well do the models generalize to NEW data?

- Since this is a toy problem, we know what the TRUE trend should be (it’s a parabola).

Test or prediction grids for visualization

- Visualizing model predictions is an important tool for interpreting and understanding model performance.

- It’s easy to create a test grid for this simple 1 input problem.



```r
test_viz <- tibble::tibble(x = seq(-2.1, 2.1, length.out = 51))
```

Test or prediction grids for visualization

- Visualizing model predictions is an important tool for interpreting and understanding model performance.

- It’s easy to create a test grid for this simple 1 input problem.

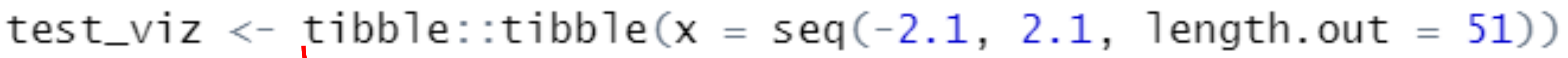
```
test_viz <- tibble::tibble(x = seq(-2.1, 2.1, length.out = 51))
```

- `seq()` function creates a SEQUENCE of evenly spaced values `from` (first argument) `to` (second argument).
- The number of values is specified by the third argument, in this case `length.out=` tells `seq()` to create 51 evenly spaced points.
- Other options exist for the third argument including `by=` which specifies the interval size between points.

Test or prediction grids for visualization

- Visualizing model predictions is an important tool for interpreting and understanding model performance.

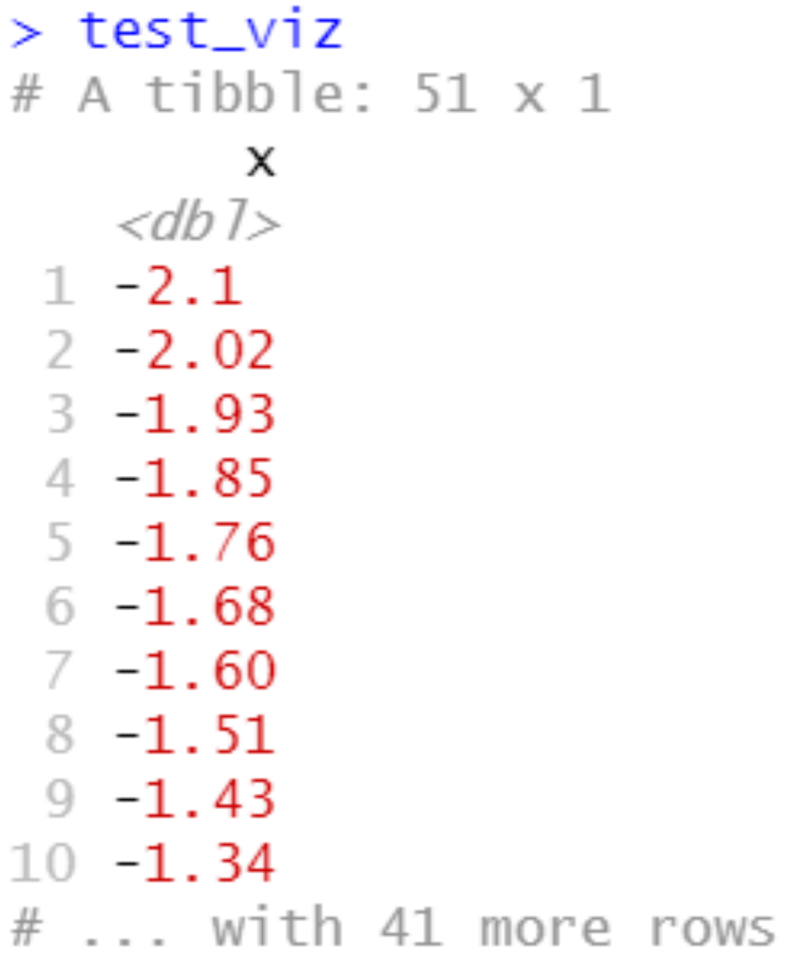
- It’s easy to create a test grid for this simple 1 input problem.



```
test_viz <- tibble::tibble(x = seq(-2.1, 2.1, length.out = 51))
```

- `seq()` creates a vector, so I assigned that vector to the named variable `x` inside a `tibble` (modern data.frame).
- This way the test grid has the SAME input name as the TRAINING SET.
- The tibble was assigned to the variable `test_viz`

Print out a few rows of the `test_viz` object



```
> test_viz
# A tibble: 51 x 1
        x
    <dbl>
 1  -2.1
 2  -2.02
 3  -1.93
 4  -1.85
 5  -1.76
 6  -1.68
 7  -1.60
 8  -1.51
 9  -1.43
10  -1.34
# ... with 41 more rows
```

- **IMPORTANT**: the test grid does NOT include output values!
- That's ok! We do NOT have to know responses to make predictions.
- We cannot compare model predictions to anything and so cannot calculate errors.
- But we can still study trends!

Making predictions in `R`

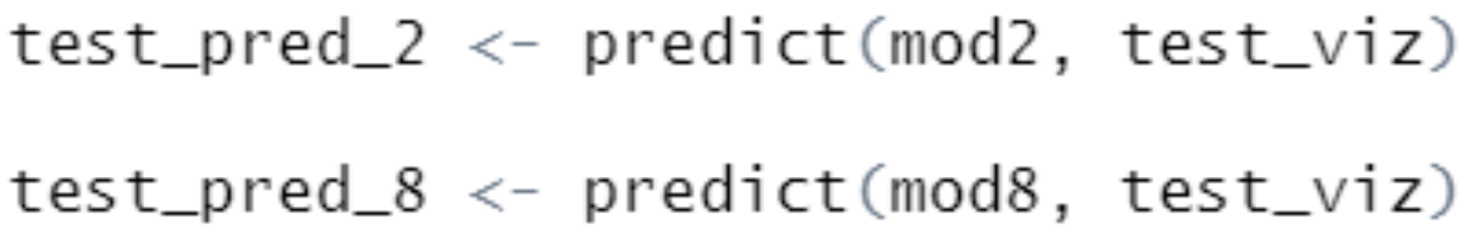
- We use the `predict()` function!

```
<variable name> <- predict(<model object>, <test set>)
```

- The <test set> could be the training set if we wanted to remake predictions on the training set later on.

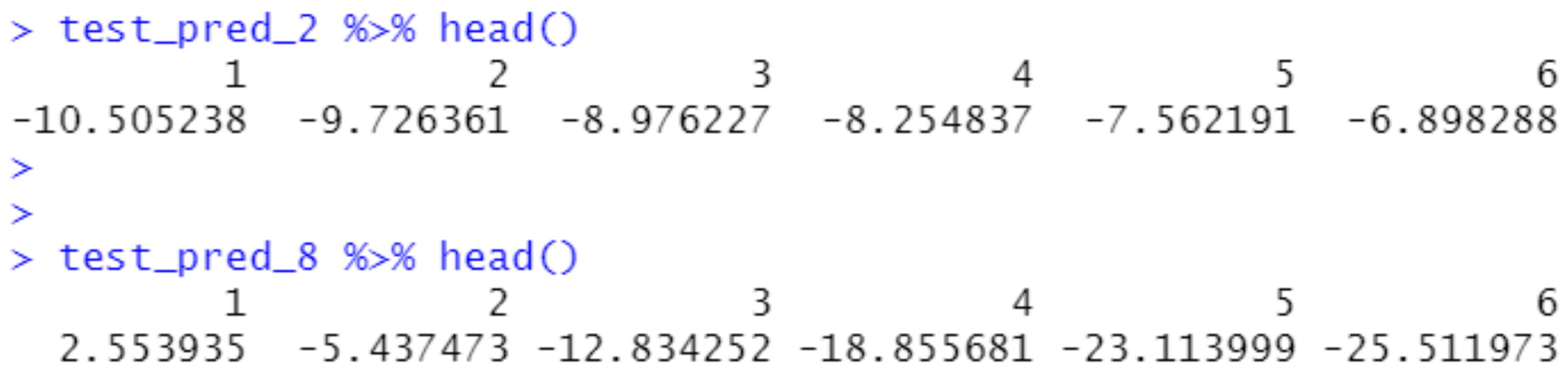
- The result assigned to <variable name> is a numeric vector.

Predictions from a few of our models



```r
test_pred_2 <- predict(mod2, test_viz)

test_pred_8 <- predict(mod8, test_viz)
```

Print out the first 6 predictions to show the numeric vector

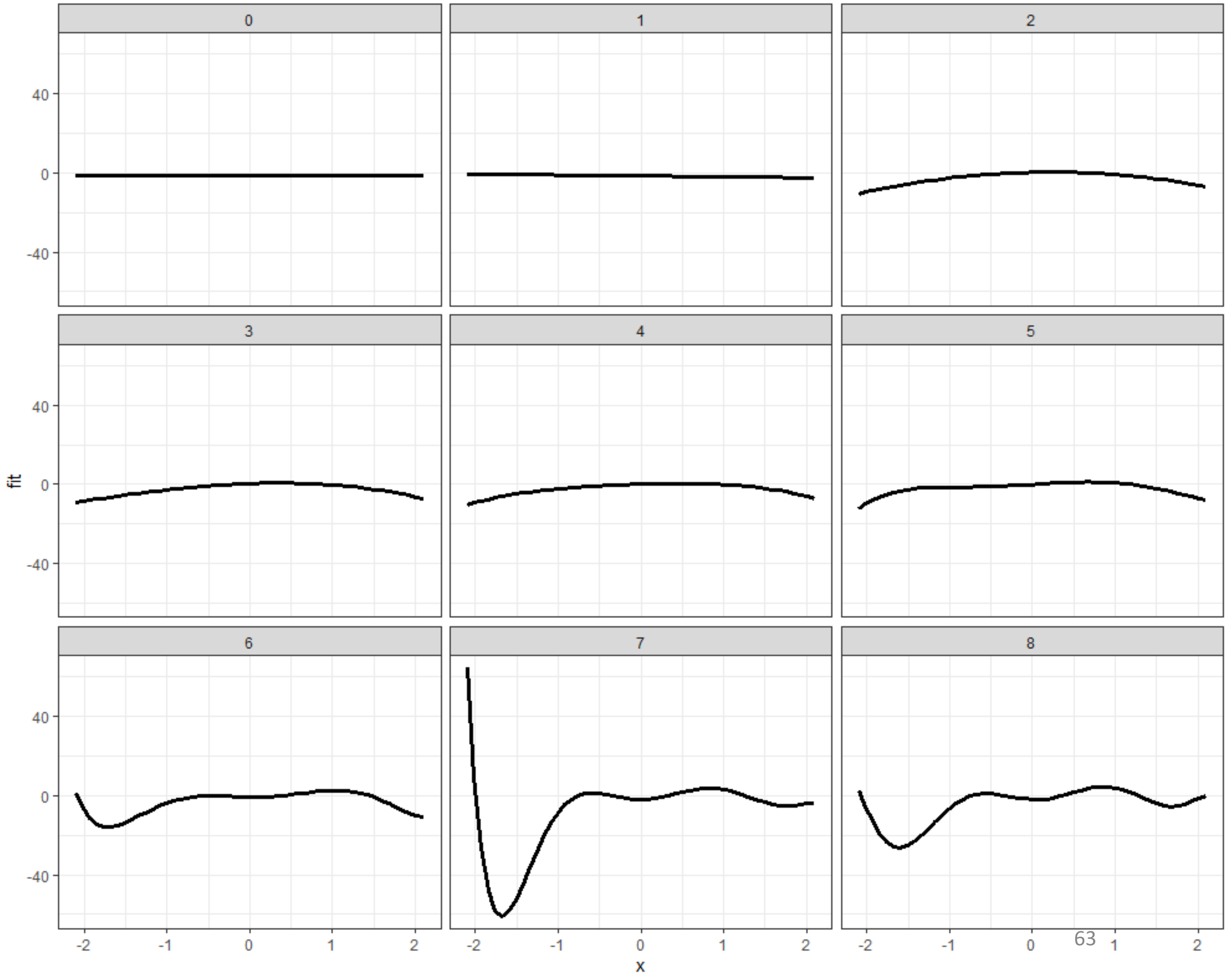


```
> test_pred_2 %>% head()
         1          2          3          4          5          6 
-10.505238  -9.726361  -8.976227  -8.254837  -7.562191  -6.898288 
>
>
> test_pred_8 %>% head()
         1          2          3          4          5          6 
  2.553935  -5.437473 -12.834252 -18.855681 -23.113999 -25.511973 
```

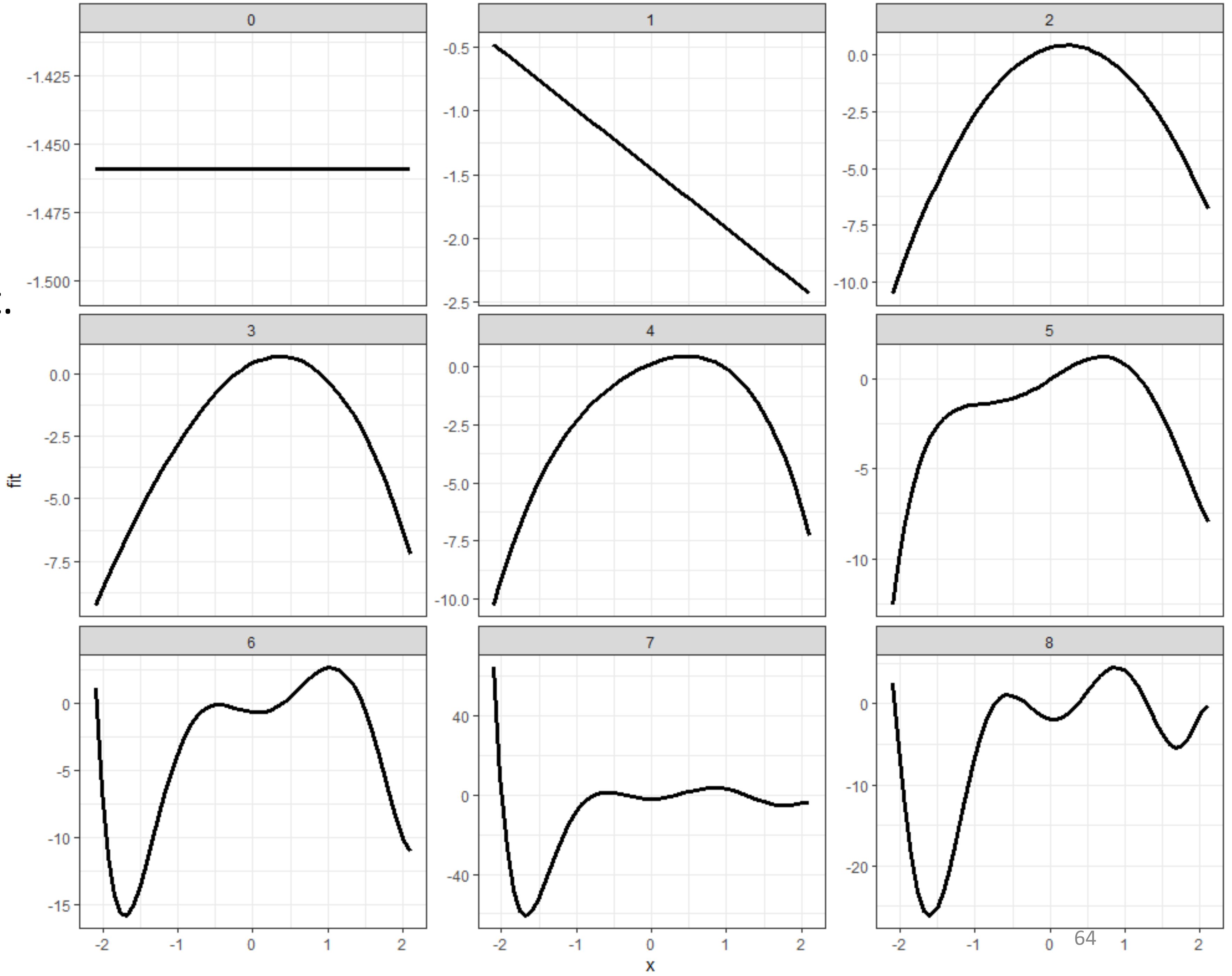
Visualize the test set predictions from all 9 models

- Next slide shows a separate facet for each model.
- The x-axis is the test set input value, `x`.
- The y-axis is the model predicted response, which is labeled `fit`.

- Y-axis scales are the SAME across all facets!
- What's clear about the trends of the higher order polynomials?



- Zoom in, by specifying the y-axis scales to be different.
  - In `facet_wrap()` specify `scales="free_y"`
- Clearly the higher degree polynomials are NOT parabolas!



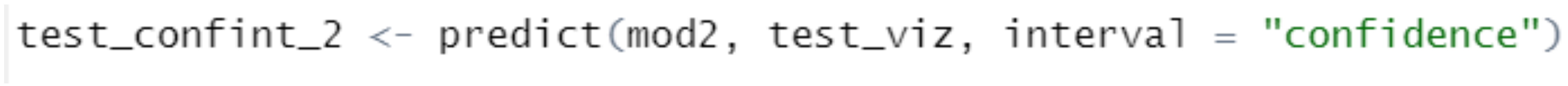
All model predictions shown so far have been the trends

- We have ignored the prediction **UNCERTAINTY**!!!

- We will spend a lot of time covering prediction uncertainty in this course.

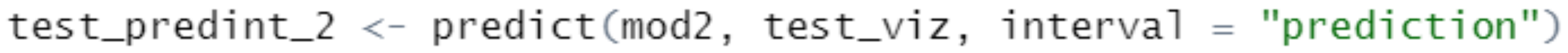
We can tell the `predict()` function to return TWO types of prediction uncertainty

- The CONFIDENCE INTERVAL



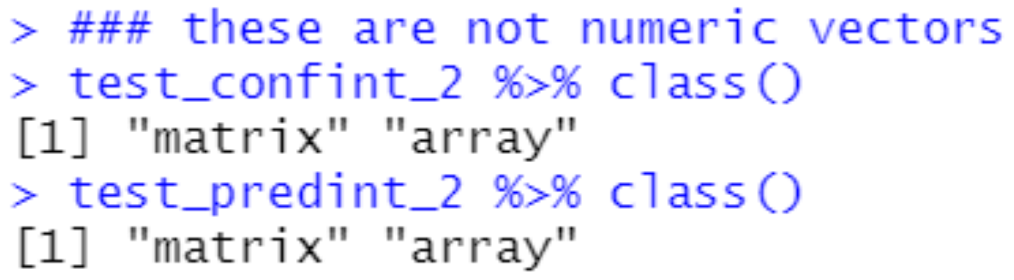
```r
test_confint_2 <- predict(mod2, test_viz, interval = "confidence")
```

- The PREDICTION INTERVAL



```r
test_predint_2 <- predict(mod2, test_viz, interval = "prediction")
```

- We will discuss what these mean in more detail later in the course.

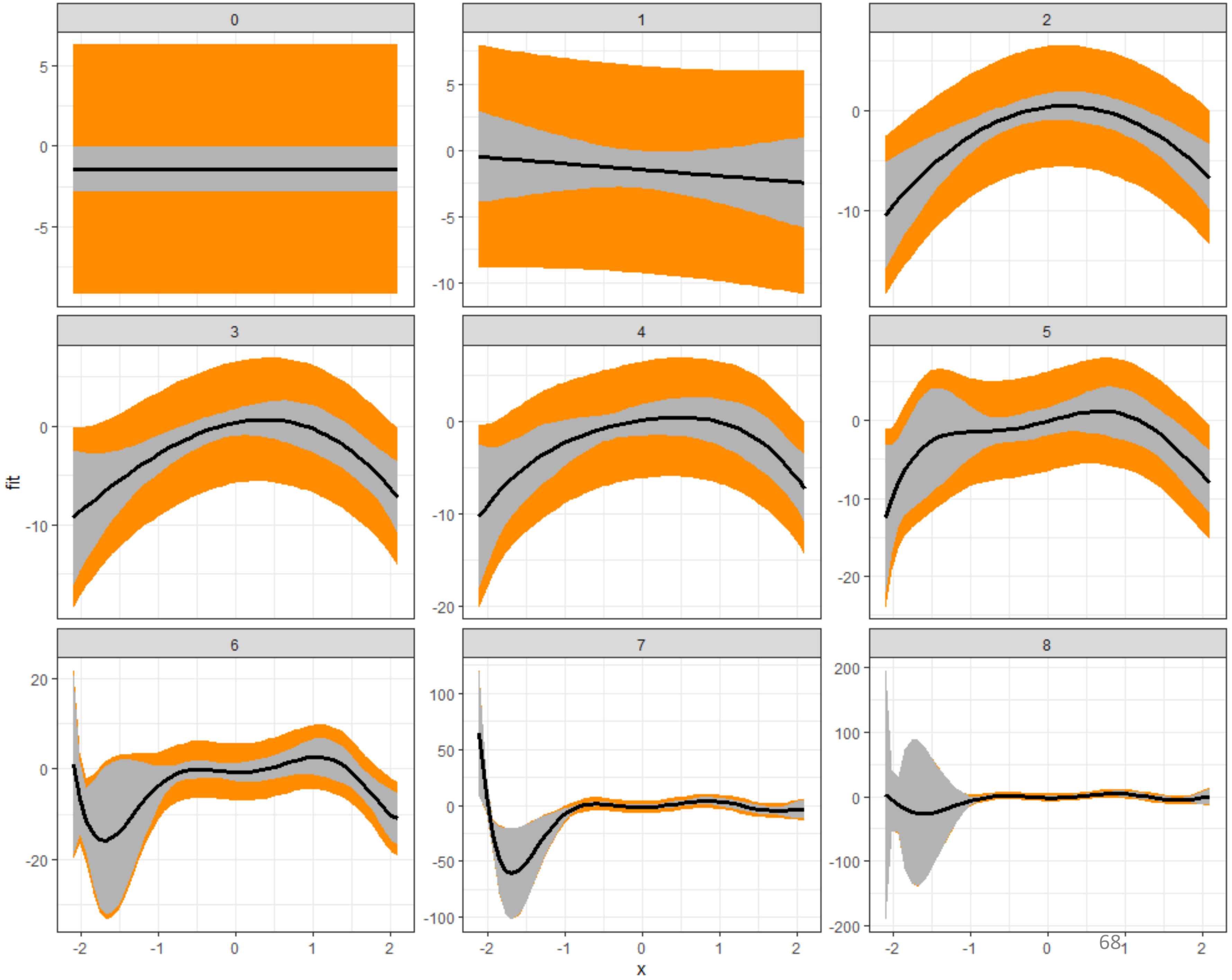


```
> ### these are not numeric vectors
> test_confint_2 %>% class()
[1] "matrix" "array"
> test_predint_2 %>% class()
[1] "matrix" "array"
```

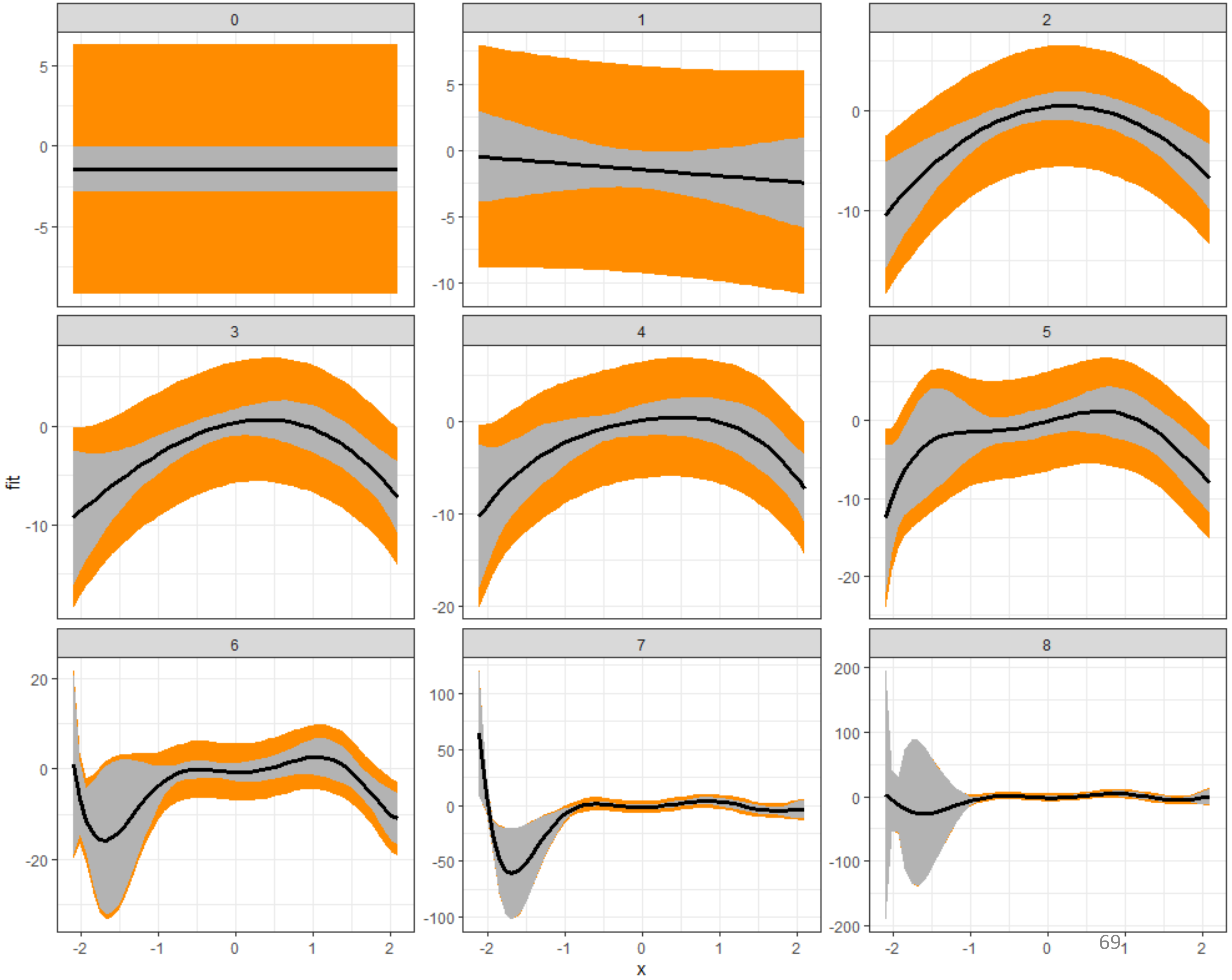
Next slide visualizes the two types of intervals

- Mean trends shown by black curves.
- Confidence intervals shown by grey ribbons around the mean.
- Prediction intervals shown by the outer most orange ribbons.

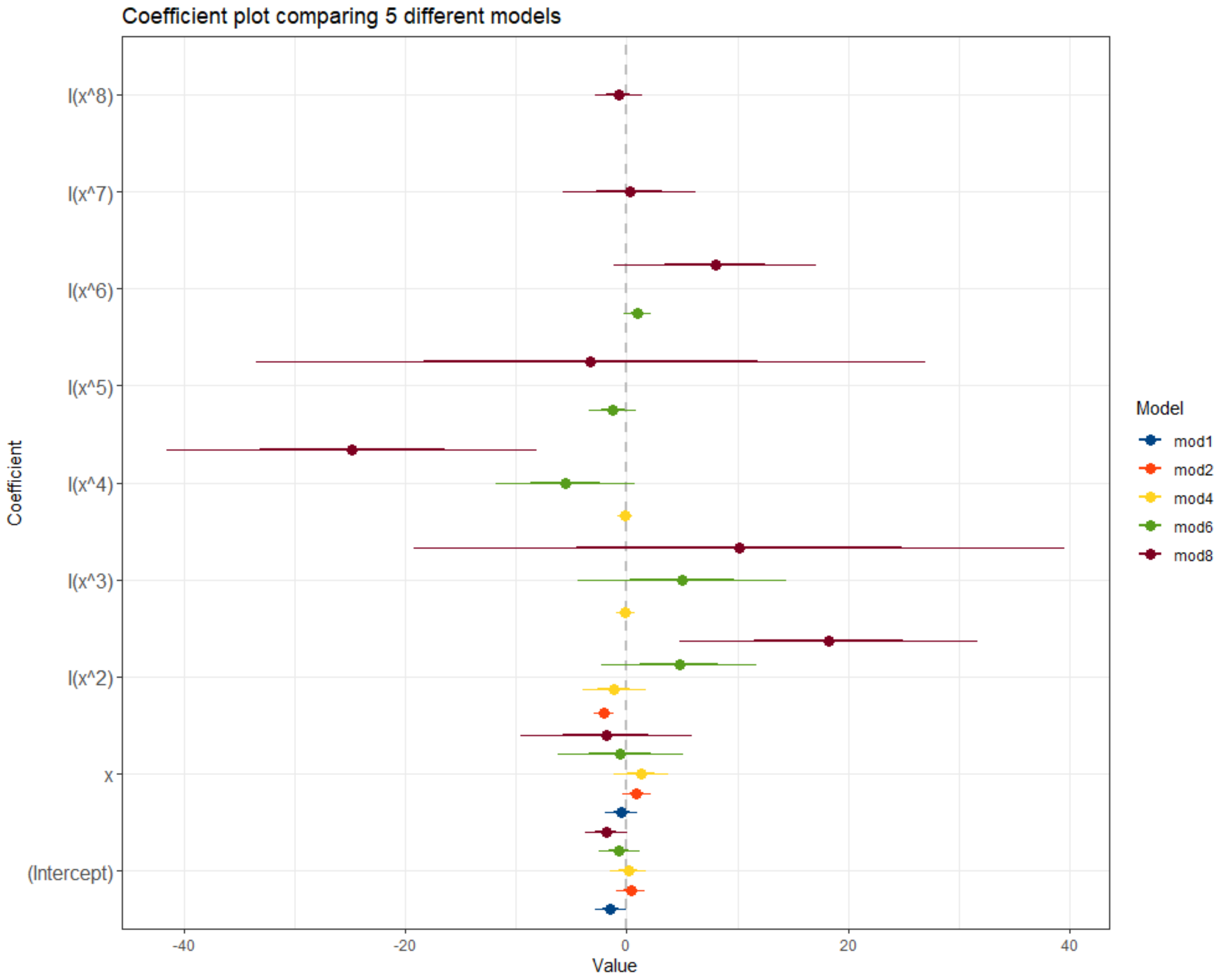
- The y-axis scales are DIFFERENT in each facet.
- Look at the range on the y-axis scale for the 7th and 8th degree polynomials!



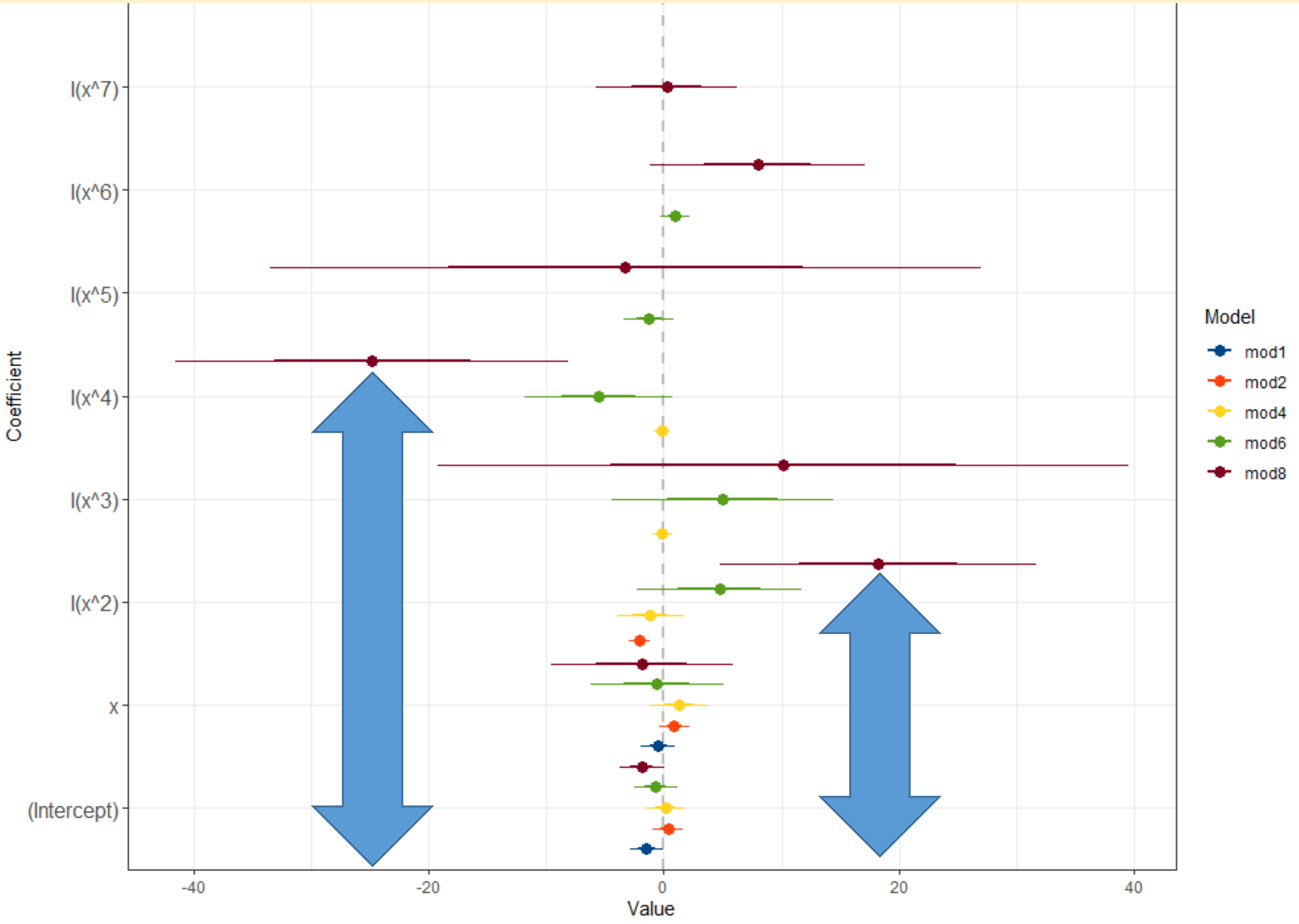
- The y-axis scales are DIFFERENT in each facet.
- Look at the range on the y-axis scale for the 7th and 8th degree polynomials!
- The confidence intervals represent that the mean trend of the higher degree polynomials are highly uncertain!
- Or we can say they are highly variable!



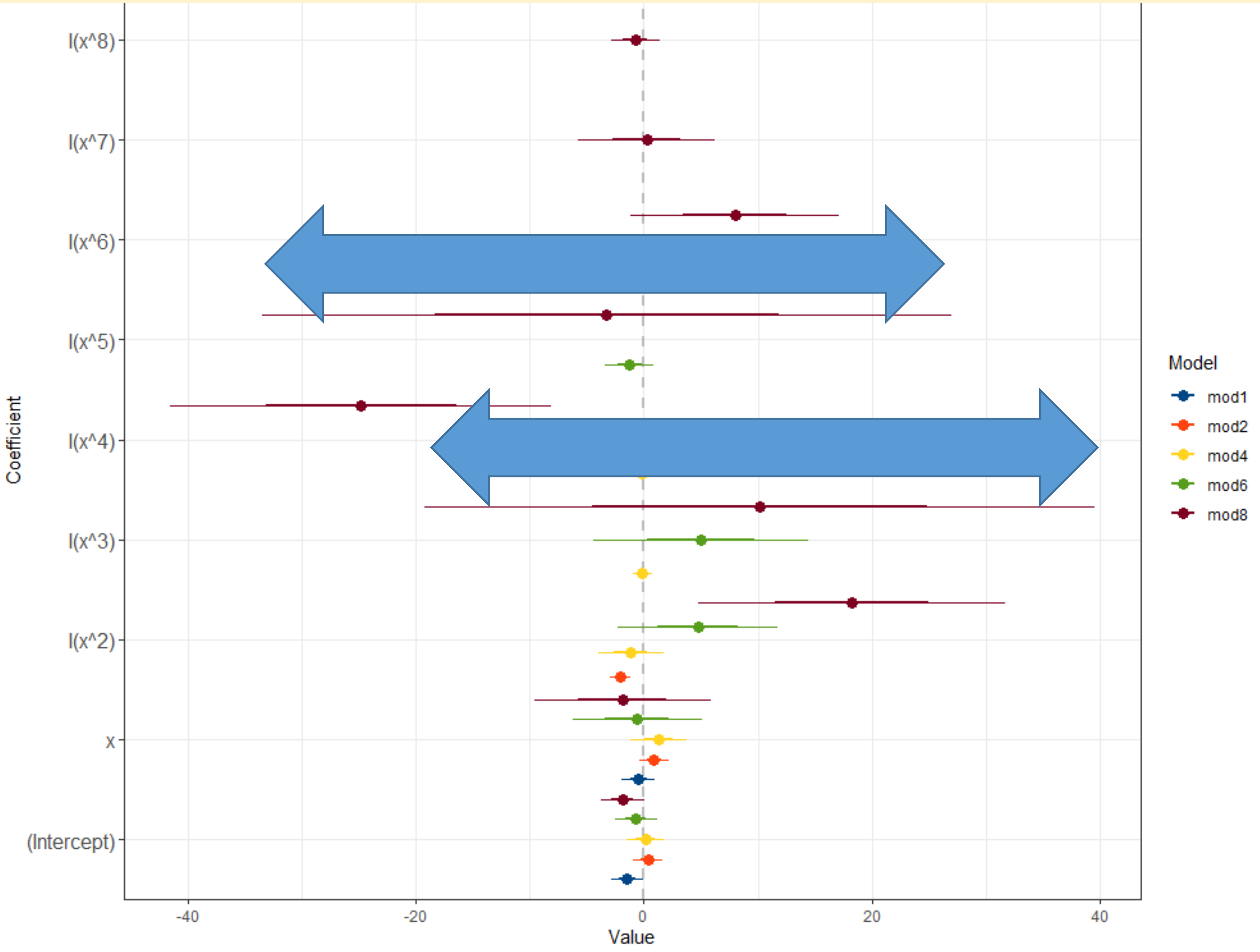
What's going on? Why is there so much variability in the higher degree polynomials?



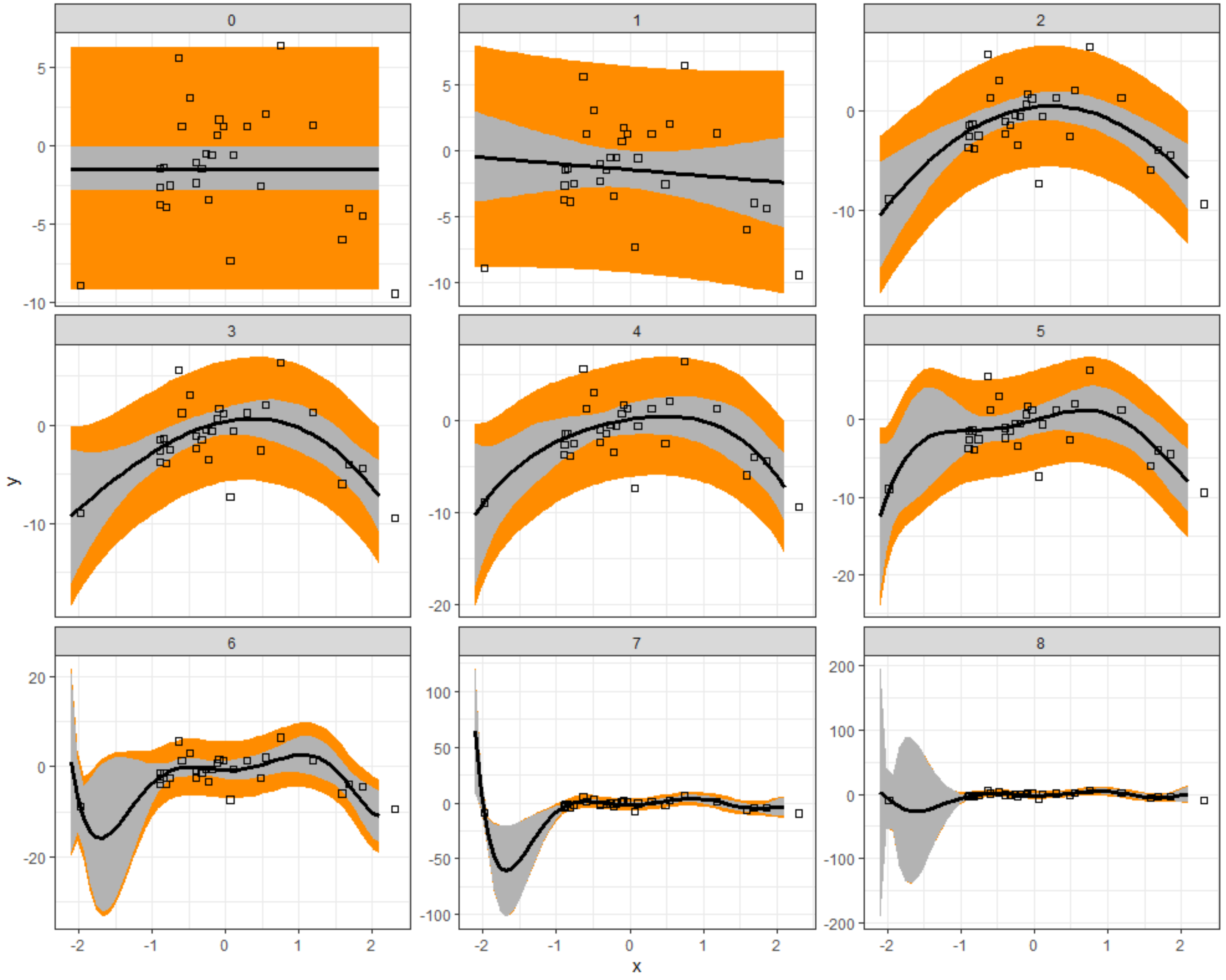
8th degree polynomial has coefficients with estimates near -20 and +20 !!!!!



8th degree polynomial has very uncertain coefficients!!



We just visualized the Bias-Variance trade-off!



In this toy demo, we knew the truth…

- In a real problem though…how can we assess our model’s behavior if the metrics on the training set tell us the wrong answer?

- We know the higher degree models are wrong because we know the true function.

In a real application, we will NOT know the truth

- If we had a new data set…that data set could represent “truth”…or rather can help us understand how our model GENERALIZES.

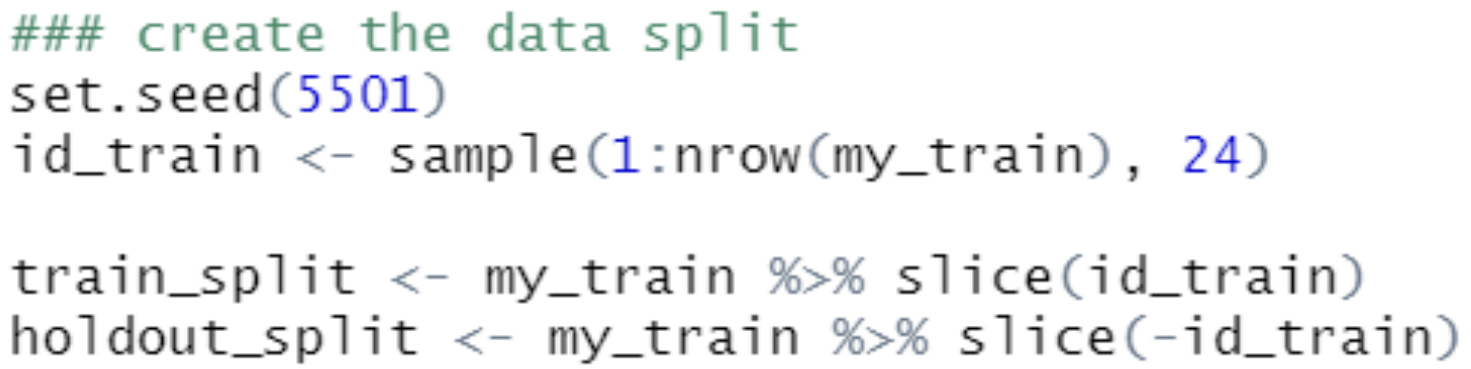
- But will we have another data set?

Data splitting to approximate “new” data!

- Break up or SPLIT the data set.
- Partition the complete data set into a dedicated TRAINING set and a dedicated HOLD-OUT test set.
- The hold-out set is used to assess the model performance.

Let’s split our data into 24 training points and 6 hold-out test points (**80/20 training/test split**)

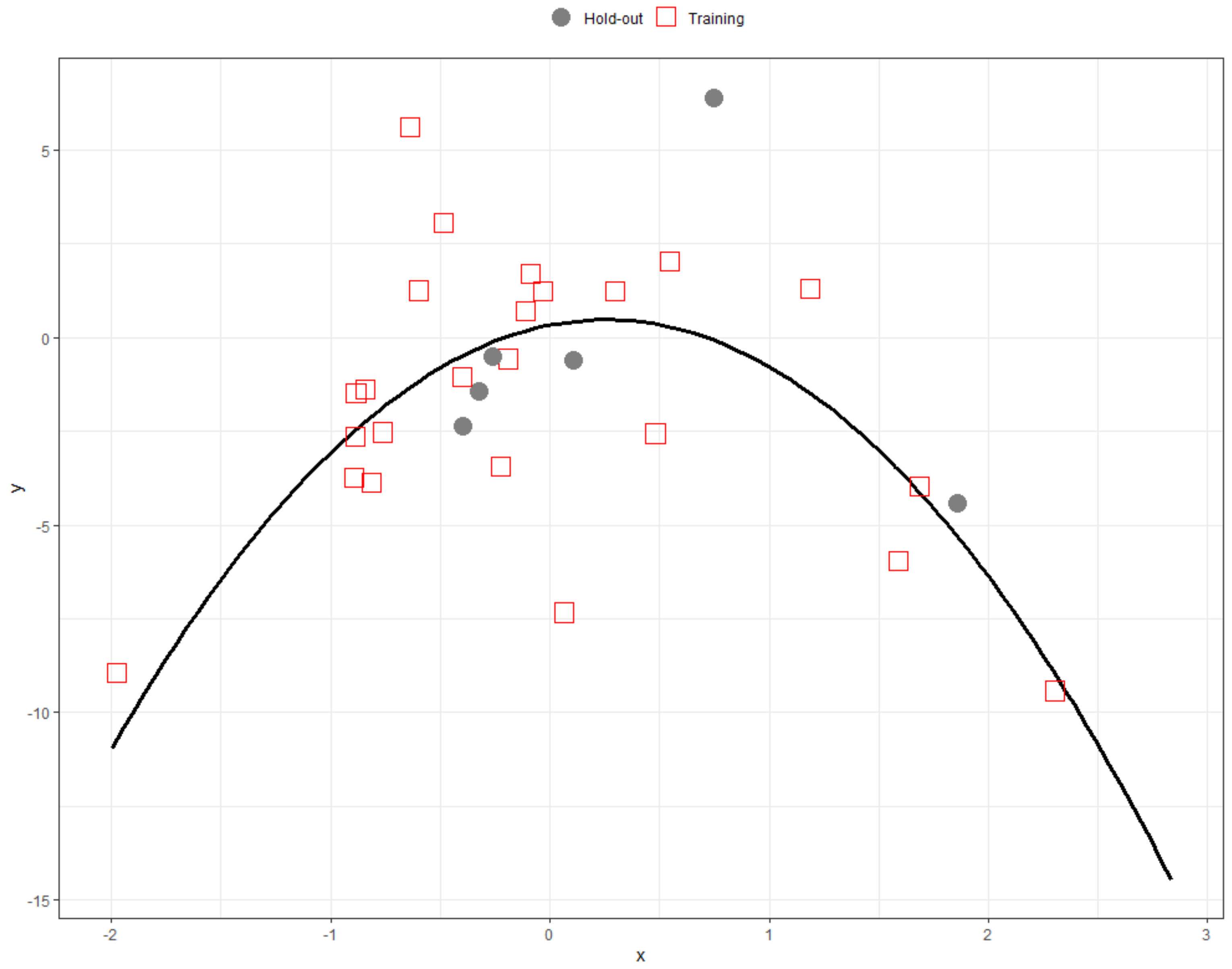
- Simple way to do this is with the `sample()` function in `R`



```
### create the data split
set.seed(5501)
id_train <- sample(1:nrow(my_train), 24)

train_split <- my_train %>% slice(id_train)
holdout_split <- my_train %>% slice(-id_train)
```

- The HOML book describes multiple ways to create data splits.

Visualize the random training and testing splits

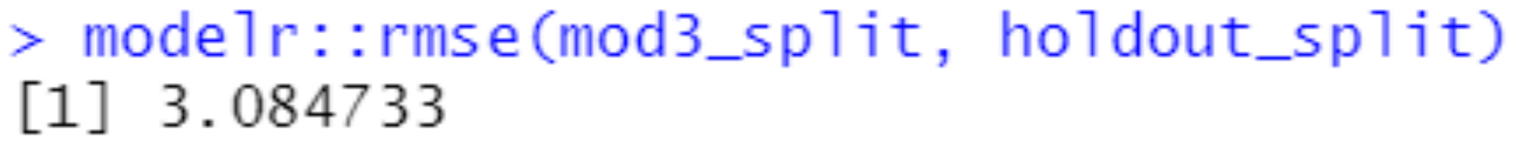


Model training and evaluation with training/test splits

- Train EACH model using just the 24 training points
  - Specify `data=train_split` to the `lm()` calls

- For example: `mod3_split <- lm(y ~ x + I(x^2) + I(x^3), data = train_split)`

- Evaluate EACH model's performance using the 6 test points, for example with the cubic relationship:

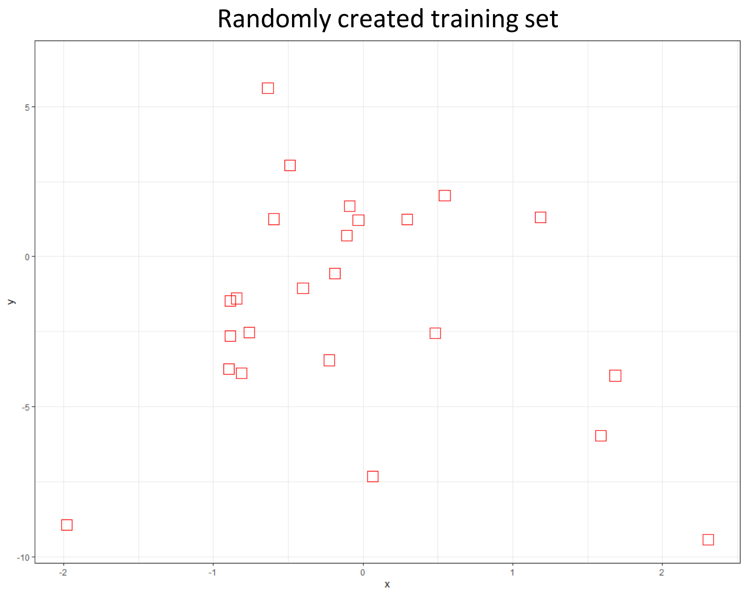


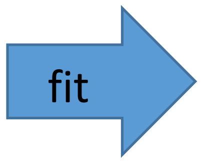
```
> modelr::rmse(mod3_split, holdout_split)
[1] 3.084733
```

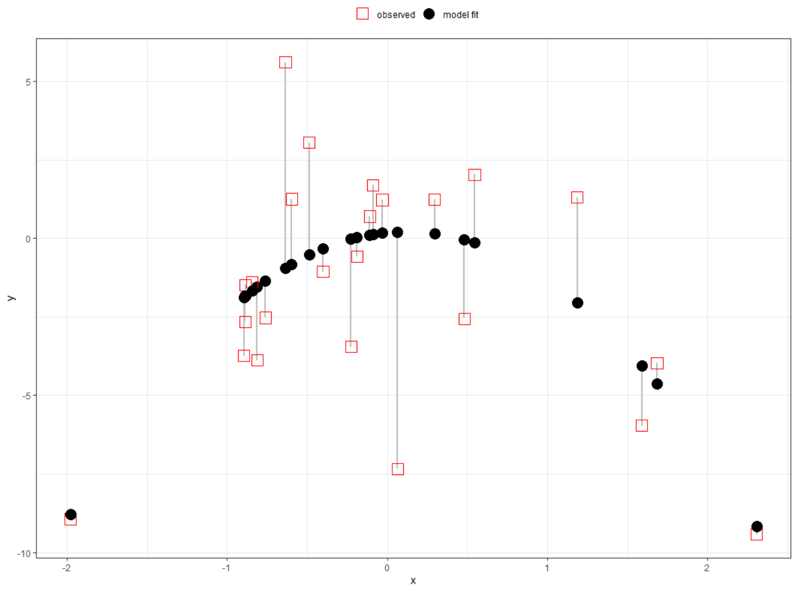
Model training and evaluation with training/test splits

- Train EACH model using just the 24 training points
  - Specify `data=train_split` to the `lm()` calls

- For example: `mod3_split <- lm(y ~ x + I(x^2) + I(x^3), data = train_split)`



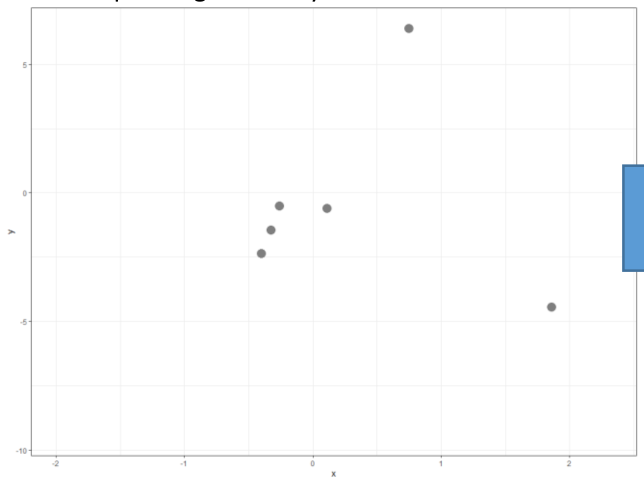




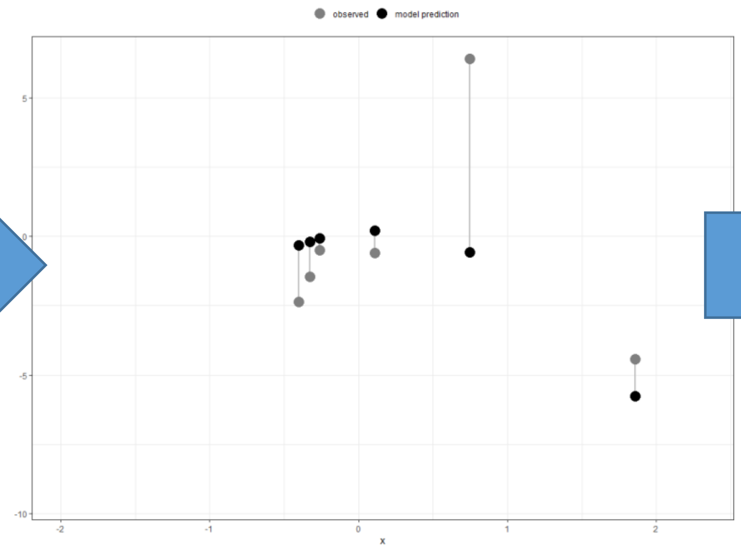
Visualizing the model fit by comparing the observed output next to model "fits" or predictions on the training set

Model training and evaluation with training/test splits

Corresponding randomly created hold-out test set



Predict hold-out test set with trained model



Summarize hold-out test set performance

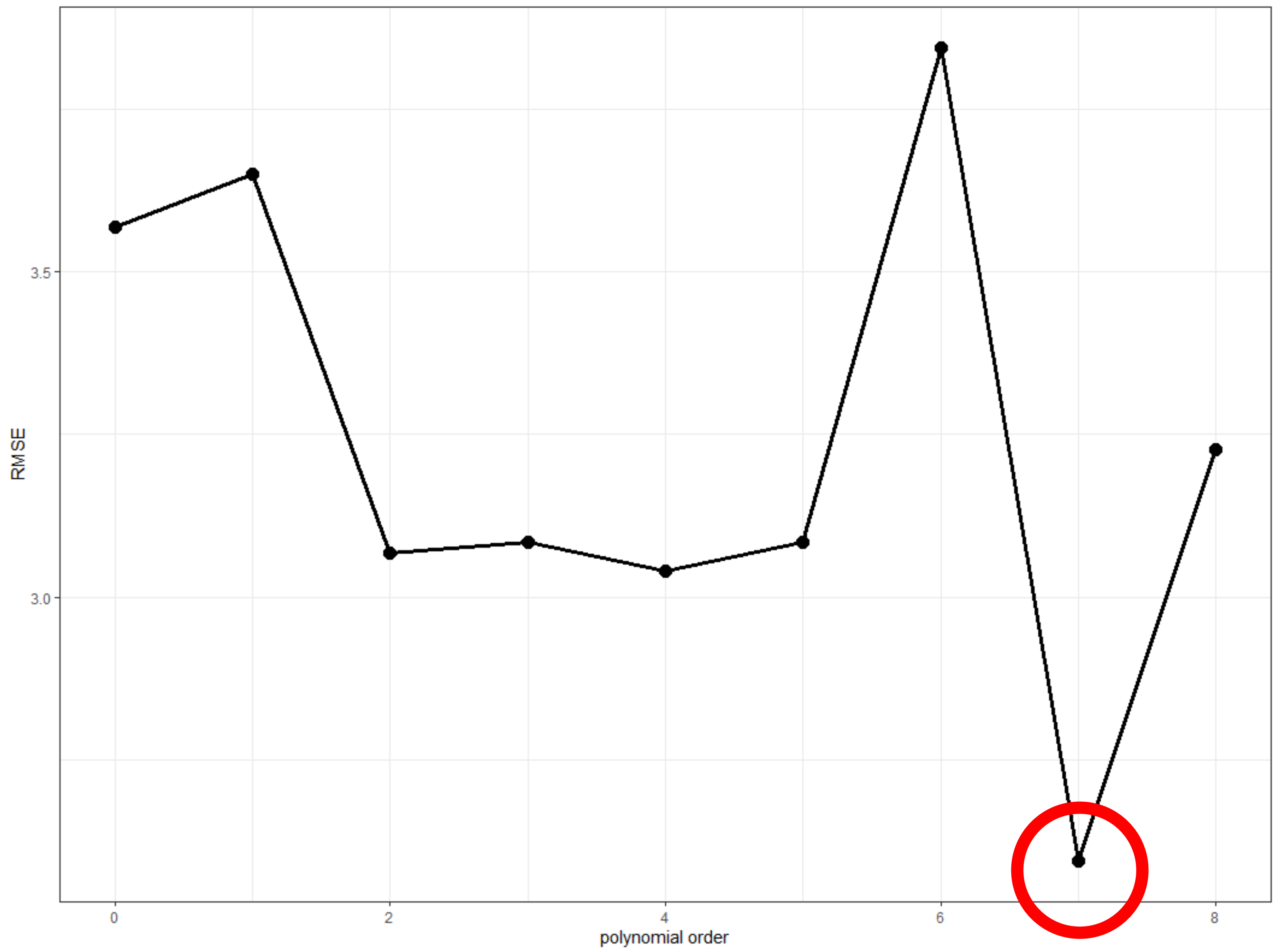
- Evaluate EACH model’s performance using the 6 test points, for example with the cubic relationship:

```
> modelr::rmse(mod3_split, holdout_split)
[1] 3.084733
```

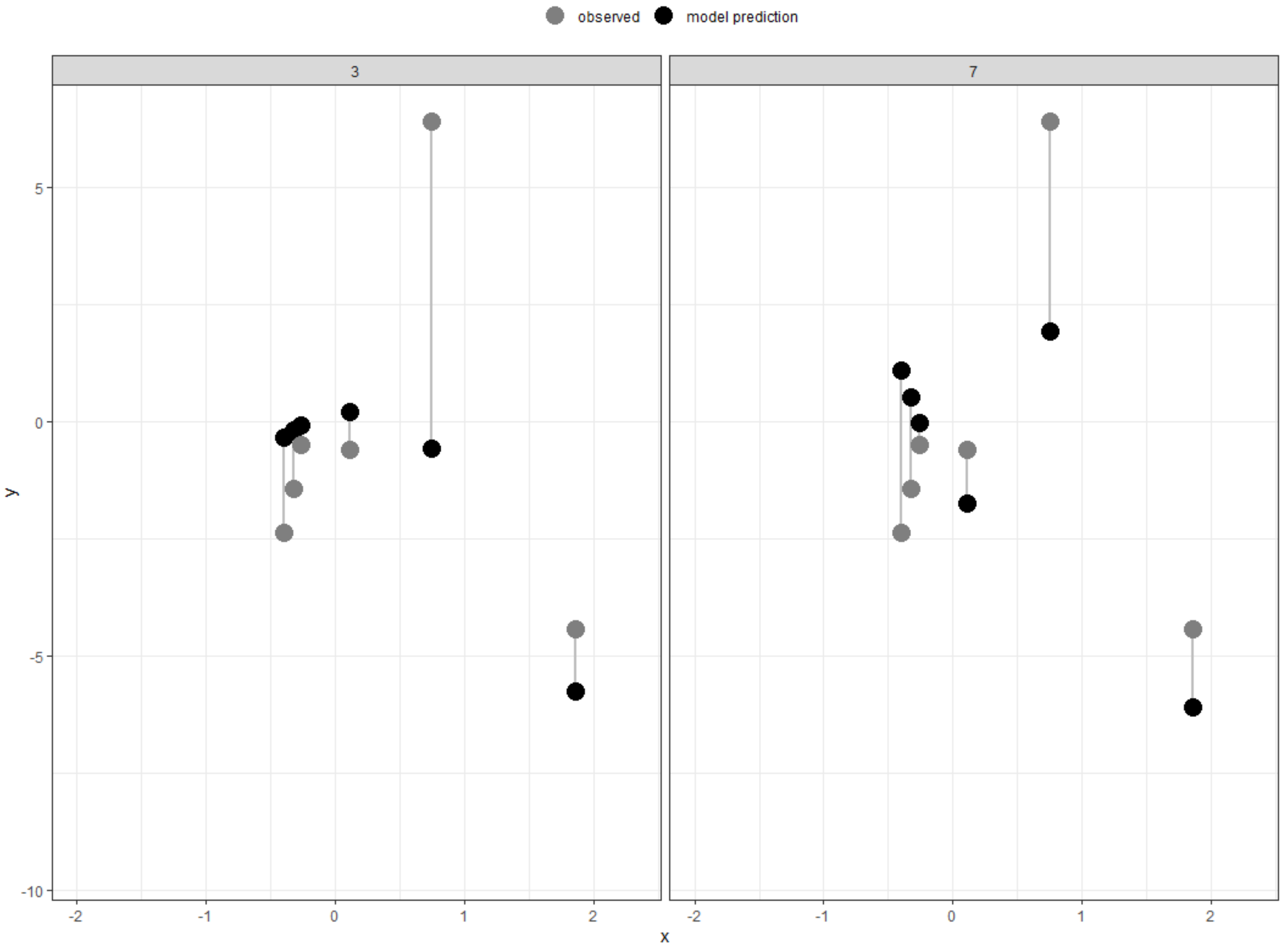
Repeat BOTH steps for every model!

- Train 9 models on the SAME training split.
- Calculate the performance metric on the SAME hold-out test split.
- Which model do you think will perform the best?

7th degree polynomial model is the best!



How?!? Shouldn't the train/test split find the right answer?!?!



Rather than splitting just once…let’s split the data multiple times

- Concept of RESAMPLING – **re**peat **sampling**
- Resample a data set to create multiple training and test splits.
- We will train EACH model multiple times and assess EACH model’s performance multiple times.

Multiple resampling techniques exist

- Two common approaches:
- Bootstrap: Resampling WITH REPLACEMENT.
- K-fold cross-validation: ensure each observation is used as a test point once.

K-fold cross-validation

- Randomly partition the data into k-FOLDS, such that each observation is in a test set once and ONLY once.

- Popular choices for k: 5-fold and 10-fold

Cross-validation has a lot of book-keeping…

- The HOML book discusses multiple libraries and functions you could use, but one approach with `modelr` is:

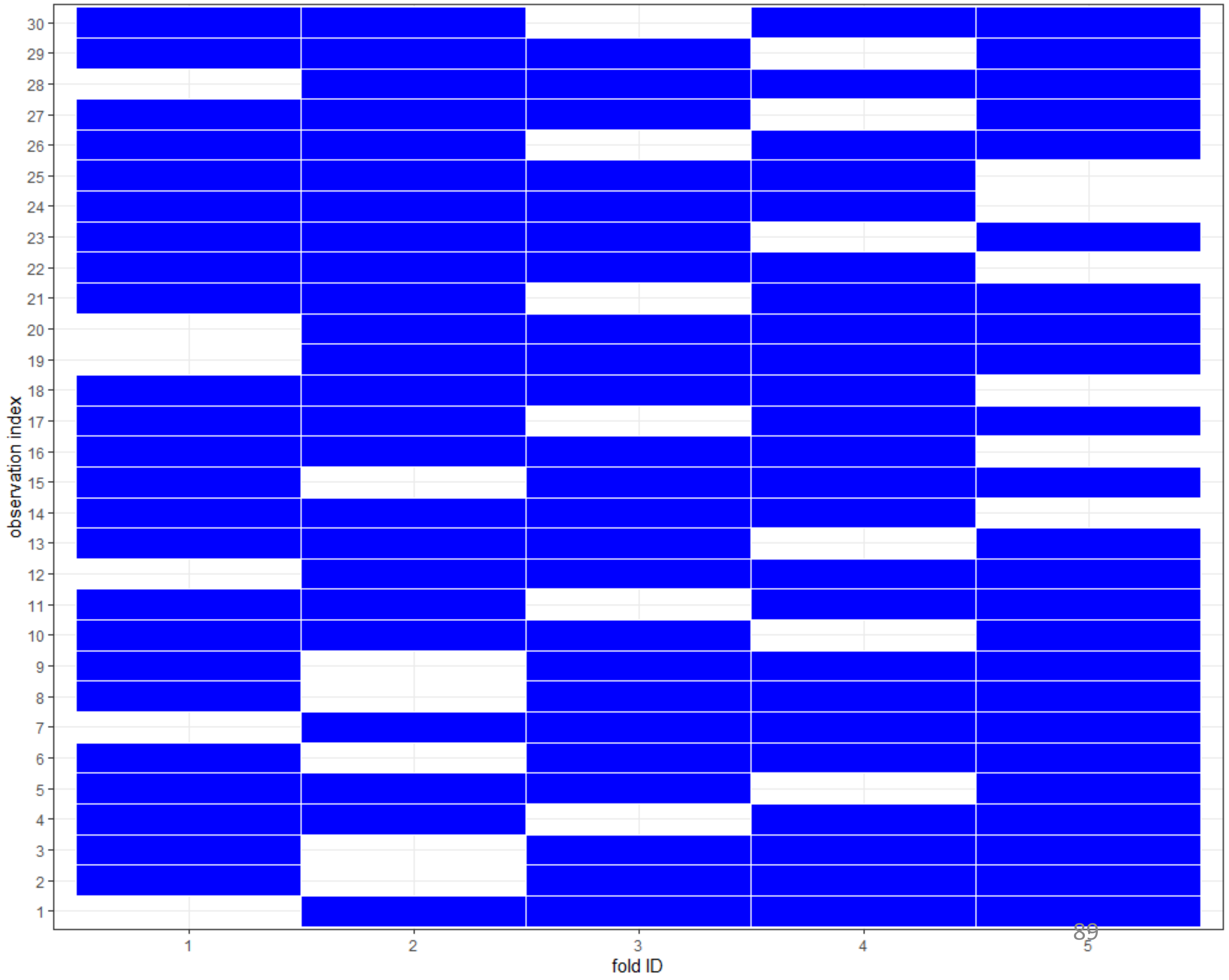
```
set.seed(23413)
cv_info_k05 <- modelr::crossv_kfold(my_train, k = 5)
```

- The resulting object is rather complex…

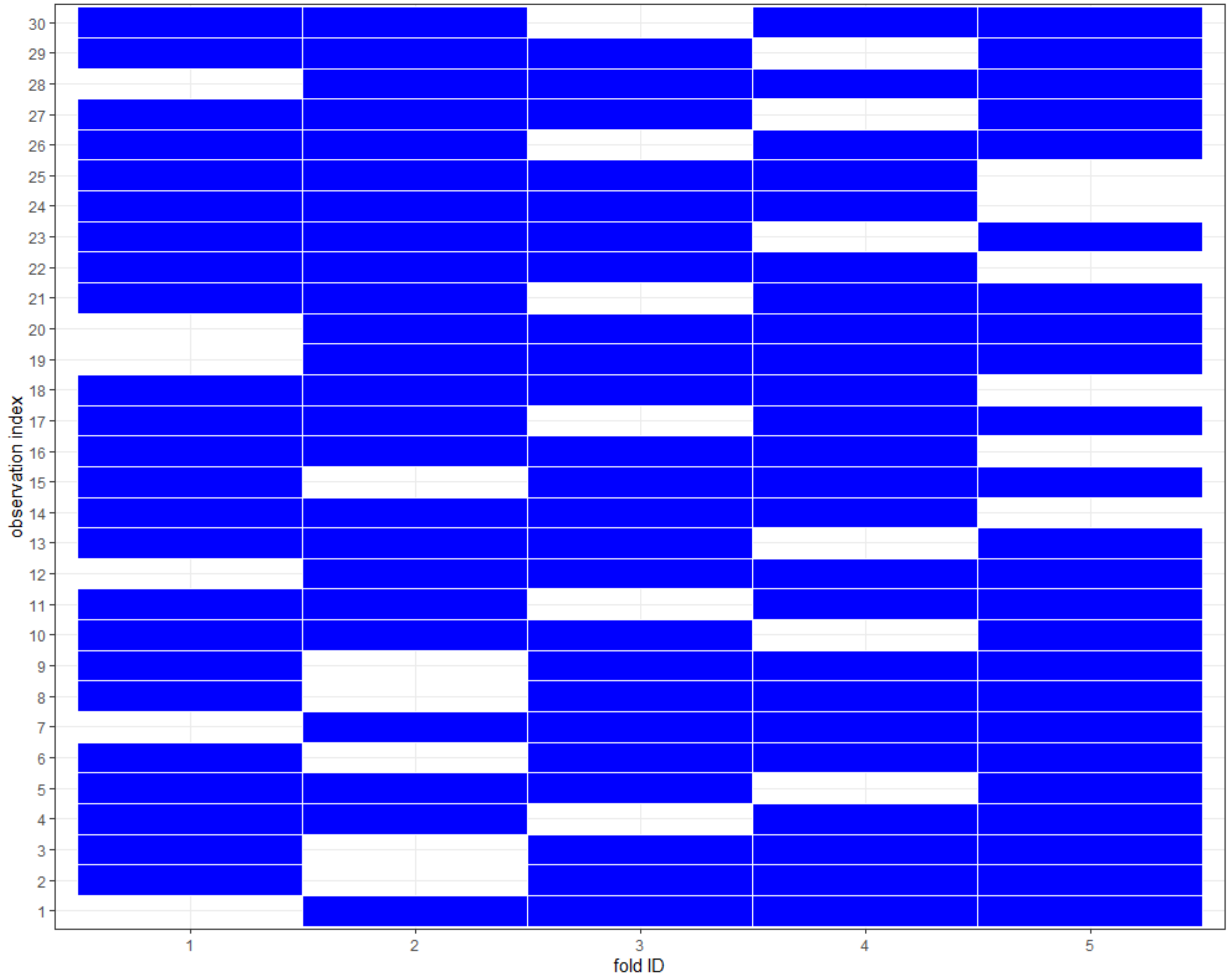
```
> cv_info_k05
# A tibble: 5 x 3
  train        test         .id  
  <named list> <named list> <chr>
1 <resample>   <resample>   1    
2 <resample>   <resample>   2    
3 <resample>   <resample>   3    
4 <resample>   <resample>   4    
5 <resample>   <resample>   5    
```

To get a sense for what's going on with k-fold cross-validation, let's visualize the ASSIGNMENTS

- Horizontal axis corresponds to the FOLD ID
- Vertical axis is an observation index.
- Blue tile denotes that an observation was randomly selected to be in the TRAINING set for that fold.
- White space represents that an observation was NOT randomly selected to be in the TRAINING set.
- **White space means the observation is HELD-OUT for the TEST set for that fold.**



How many times is an observation in a TEST set across all folds?



Steps in 5-fold cross-validation

- Start with fold 01.

- Train the model using fold 01’s training split:

```r
mod2_cv5_F01 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[1]]))
```

- Calculate the model’s performance metric using fold 01’s test split:

```r
rmse2_cv5_F01 <- modelr::rmse(mod2_cv5_F01,
                              as.data.frame(cv_info_k05$test[[1]]))
```

REPEAT for the other folds!

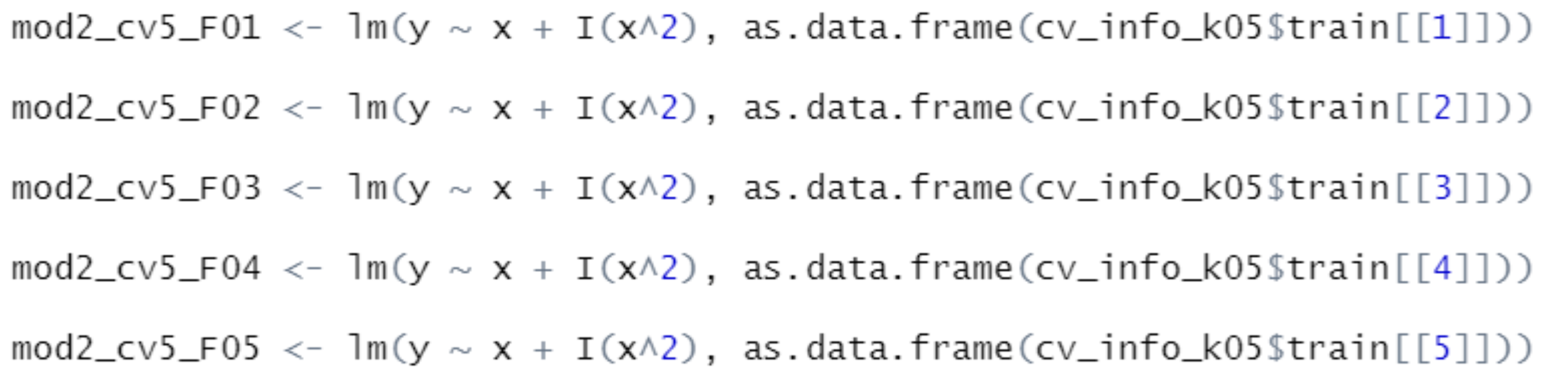
- IMPORTANT: do NOT cross the FOLDS!!!!
- Fold 01: train with fold 01’s training split. Assess with fold 01’s test split.
- Fold 02: train with fold 02’s training split: Assess with fold 02’s test split.
- Etc…

Each model therefore gets trained 5 times and tested 5 times

- Combine all fold hold-out test set performance metrics into a vector.
- Summarize the model performance across the folds by averaging!
- The fold AVERAGED performance metric represents the EXPECTED model behavior on new data!

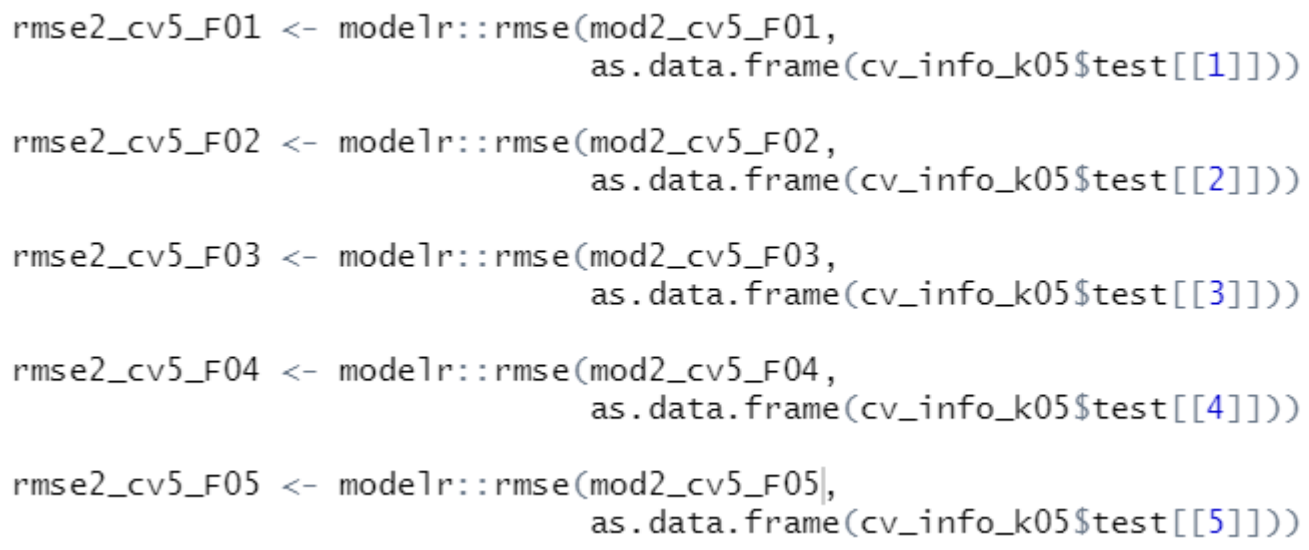
For this specific example

Train the model in each fold



```r
mod2_cv5_F01 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[1]]))

mod2_cv5_F02 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[2]]))

mod2_cv5_F03 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[3]]))

mod2_cv5_F04 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[4]]))

mod2_cv5_F05 <- lm(y ~ x + I(x^2), as.data.frame(cv_info_k05$train[[5]]))
```

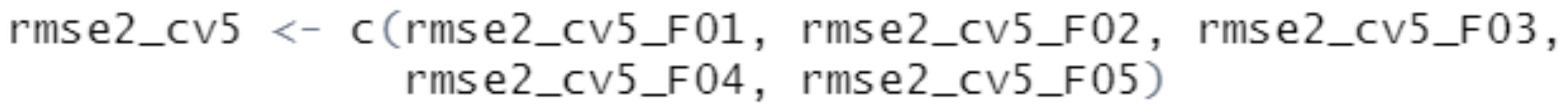
Test the model in each fold



```r
rmse2_cv5_F01 <- modelr::rmse(mod2_cv5_F01,
                              as.data.frame(cv_info_k05$test[[1]]))

rmse2_cv5_F02 <- modelr::rmse(mod2_cv5_F02,
                              as.data.frame(cv_info_k05$test[[2]]))

rmse2_cv5_F03 <- modelr::rmse(mod2_cv5_F03,
                              as.data.frame(cv_info_k05$test[[3]]))

rmse2_cv5_F04 <- modelr::rmse(mod2_cv5_F04,
                              as.data.frame(cv_info_k05$test[[4]]))

rmse2_cv5_F05 <- modelr::rmse(mod2_cv5_F05,
                              as.data.frame(cv_info_k05$test[[5]]))
```

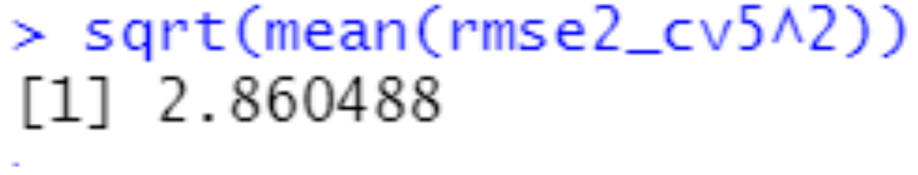
For this specific example

<u>Combine all the fold RMSEs together</u>



```
rmse2_cv5 <- c(rmse2_cv5_F01, rmse2_cv5_F02, rmse2_cv5_F03,
               rmse2_cv5_F04, rmse2_cv5_F05)
```

<u>Calculate the average!</u>



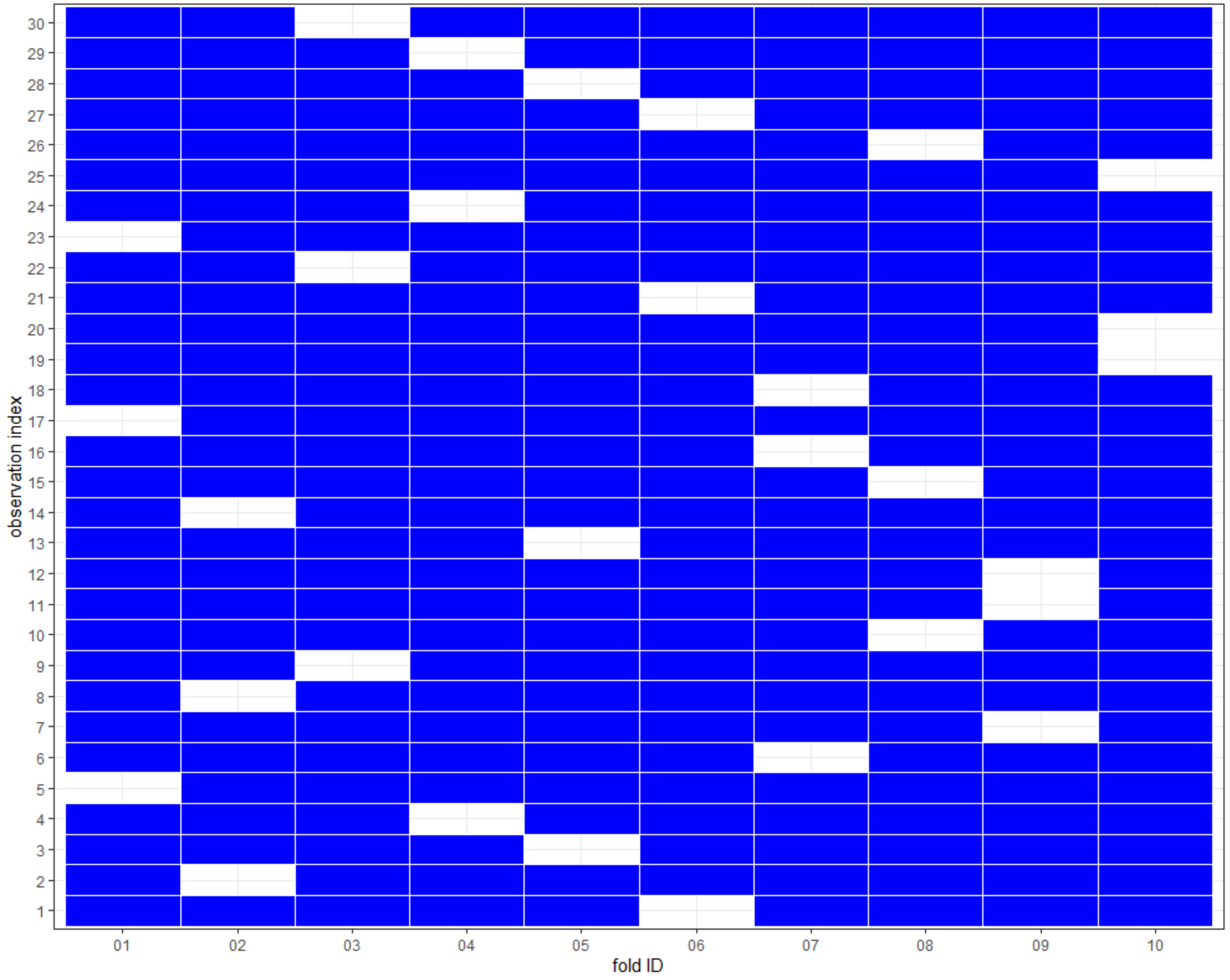
```
> sqrt(mean(rmse2_cv5^2))
[1] 2.860488
```

But we did not have to just use 5-folds!

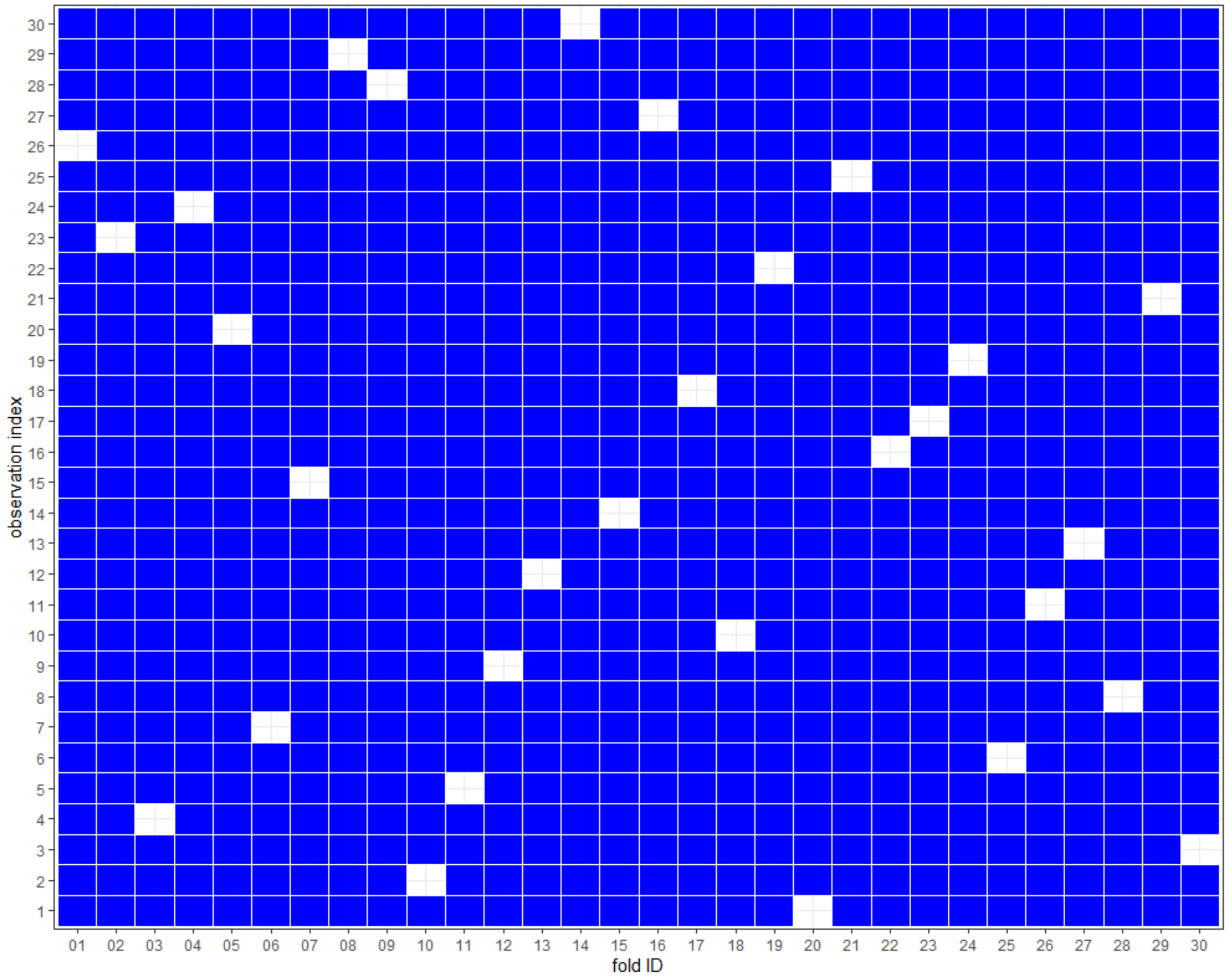
- What would it look like if we used 10-fold CV?

- Or even more folds!

10-fold cross-validation



Continue all the way to Leave-One-Out (LOO) cross-validation



How many folds should we use?

- Smaller number of folds, such as 5-fold CV, have more points in each test set.
  - **May lead to less variation in the error estimates across the folds.**
  - **Fewer folds used to average performance**!

- Using more folds decreases the number of points in each test set.
  - LOO has 1 point in each test set!
  - **Will have greater variation in the error estimates across the folds.**
  - **More folds used to average performance**!

How many folds should we use?

- Ultimately, the choice is driven by run times.

- If a model takes 1 hour to train and there are 10^5 data points…unfortunately LOOCV will be impractical…

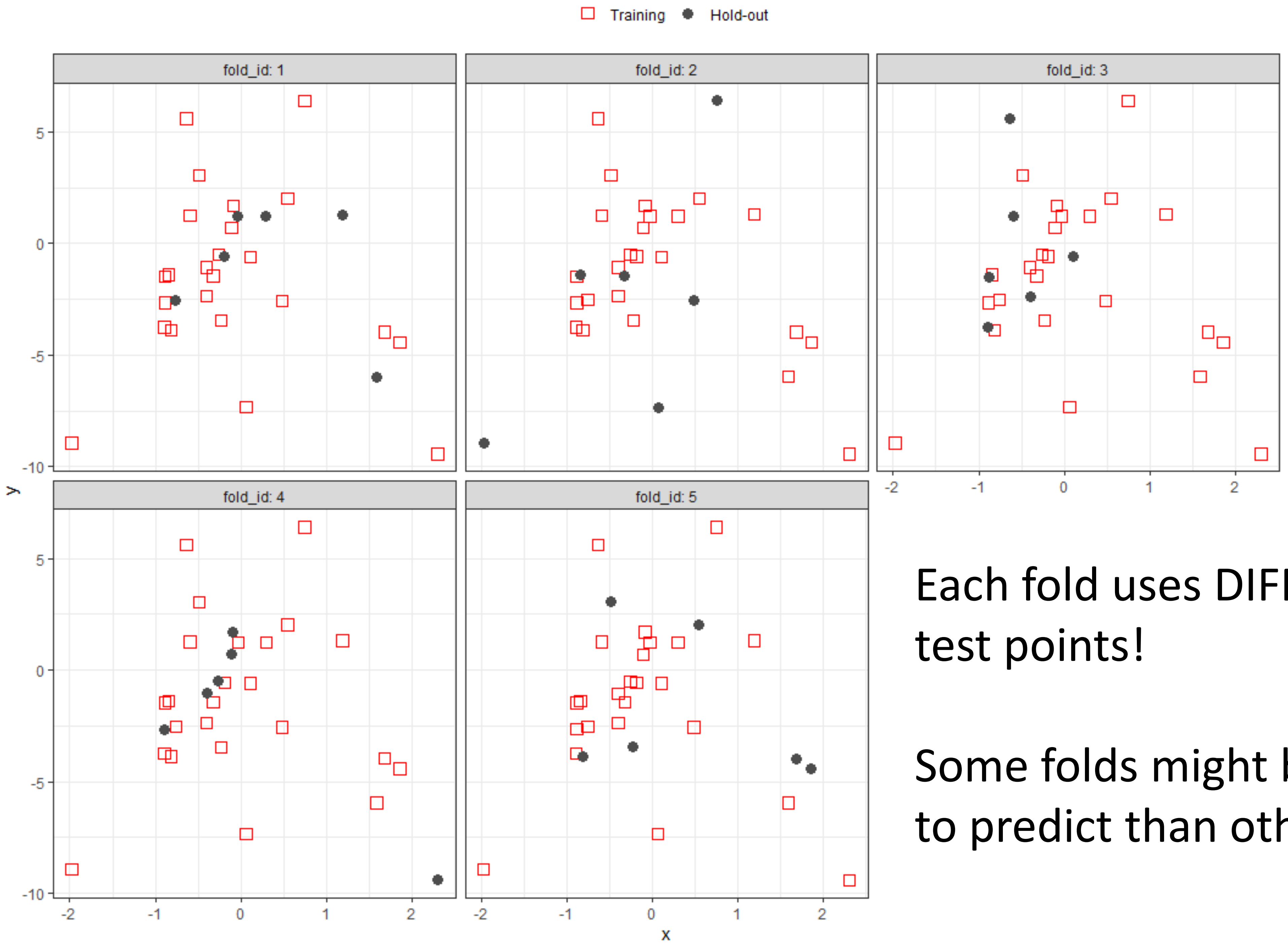
- Small numbers of folds are popular really for this reason.

- Repeated k-fold cross-validation is useful for trying the small number of folds multiple times!

We already performed 5-fold cross-validation on the quadratic relationship

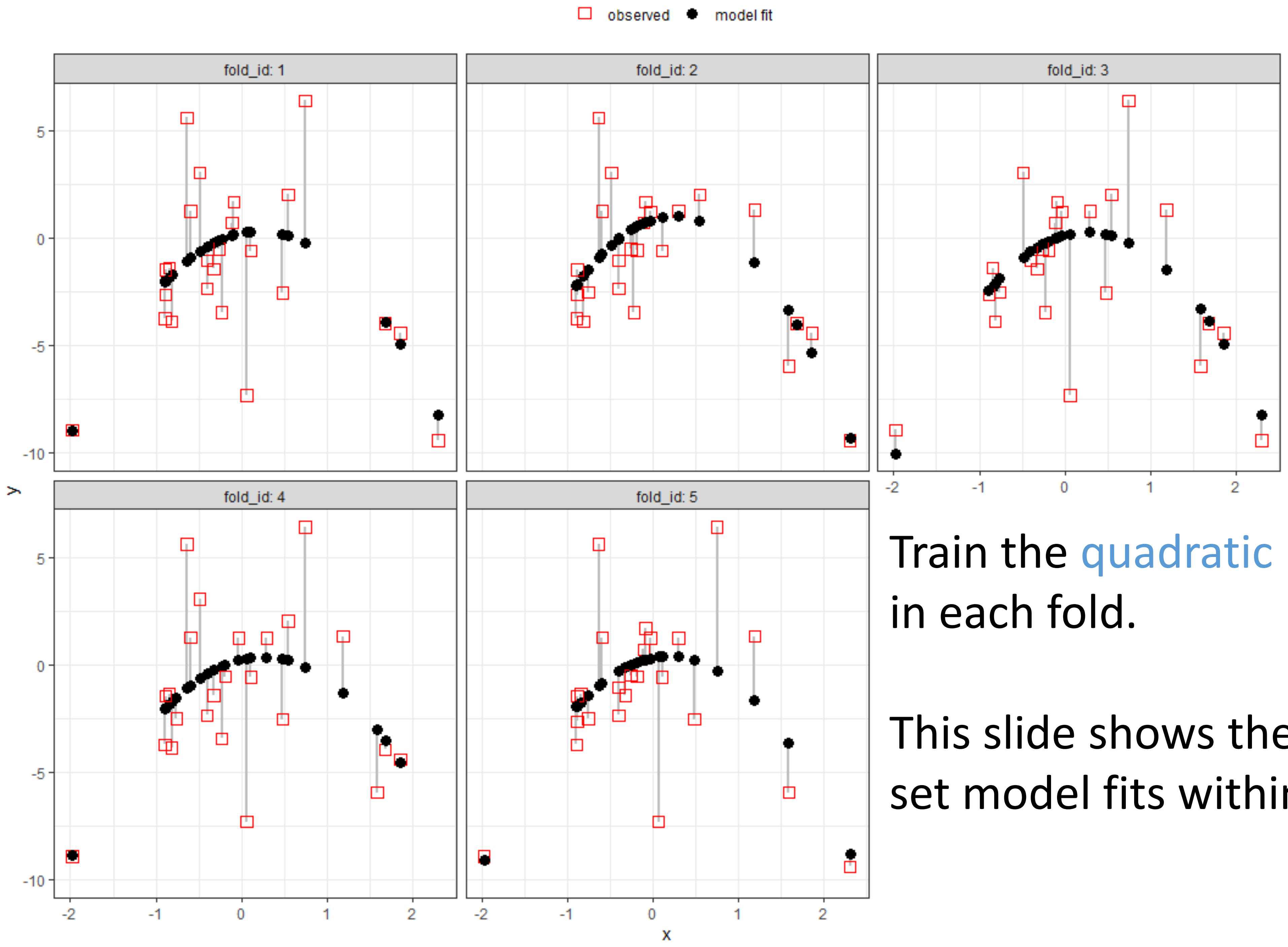
- But let’s visualize the quadratic model behavior within each fold.

- Then we will look at the model behavior for the 7th degree polynomial and see how that compares to the quadratic relationship.



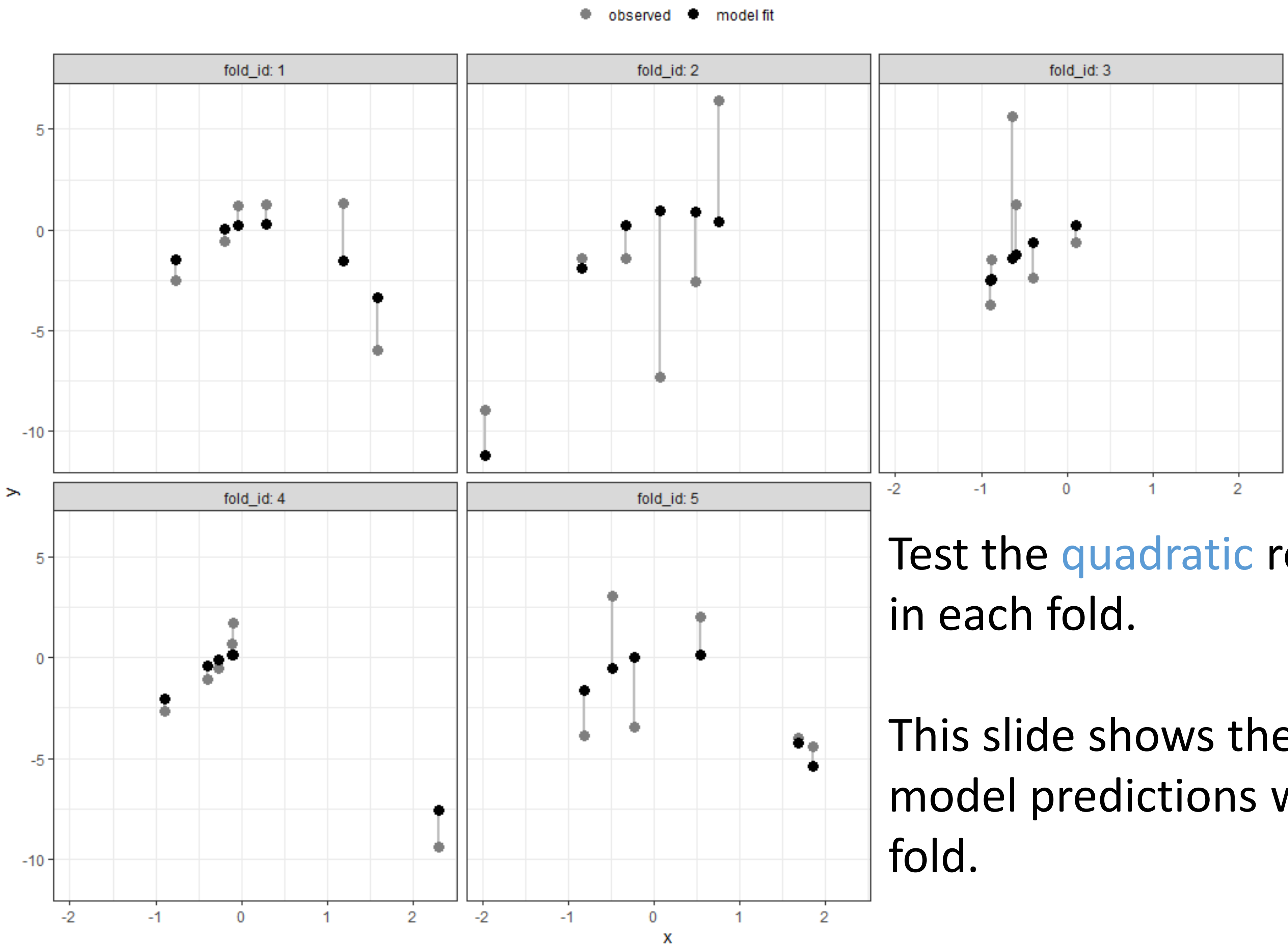
Each fold uses DIFFERENT test points!

Some folds might be “easier” to predict than others!



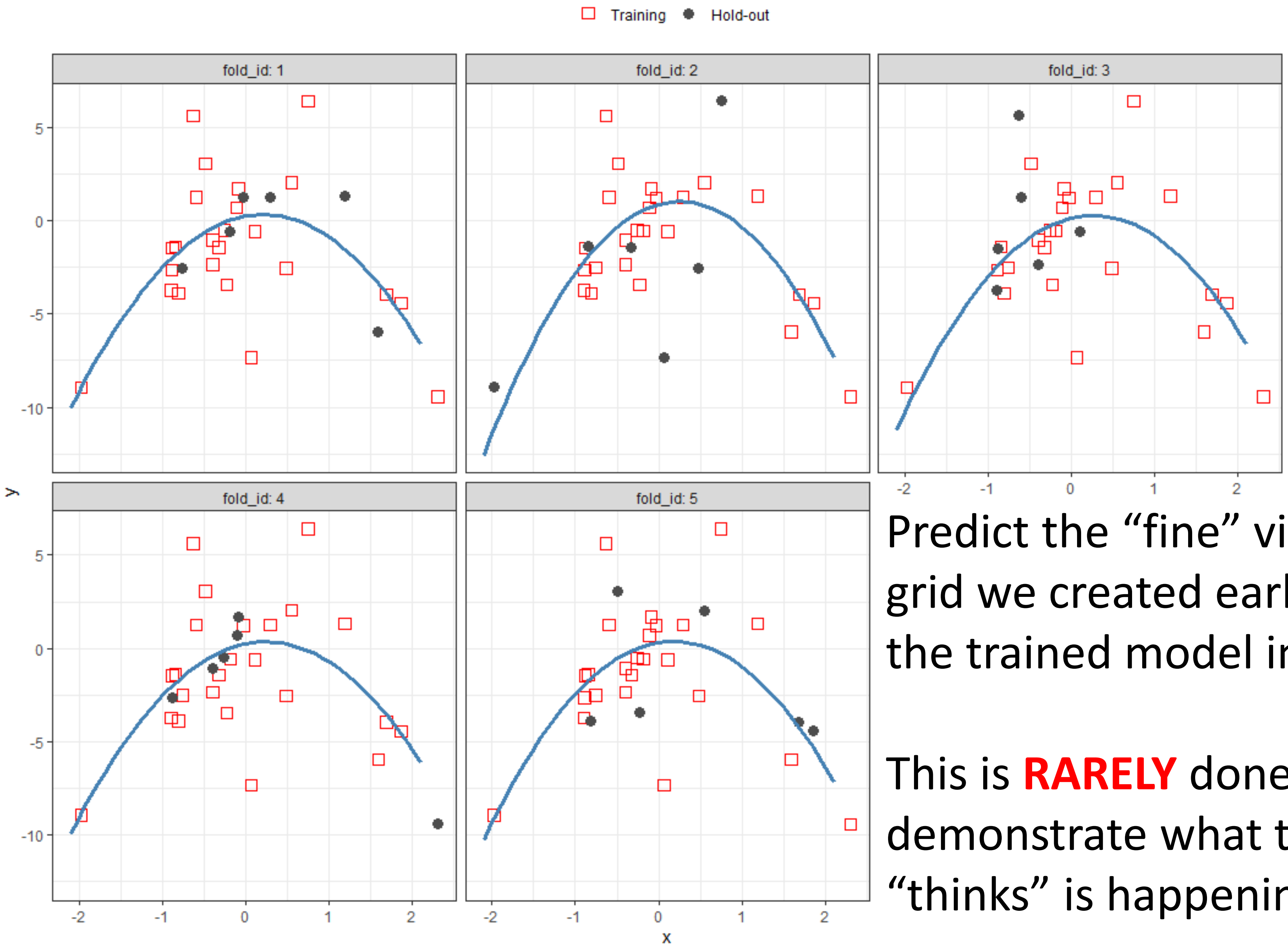
Train the quadratic relationship in each fold.

This slide shows the training set model fits within each fold.



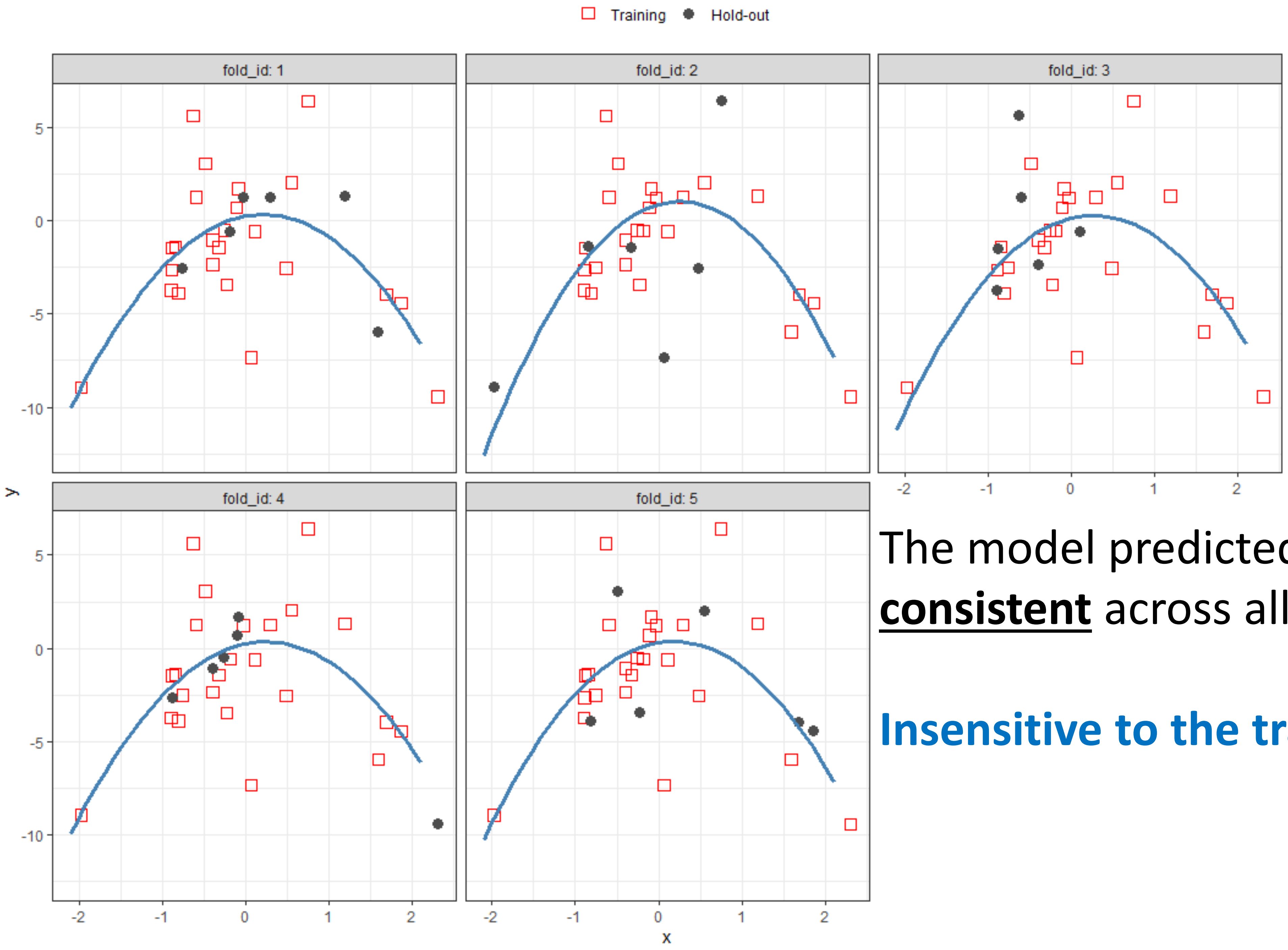
Test the quadratic relationship in each fold.

This slide shows the test set model predictions within each fold.



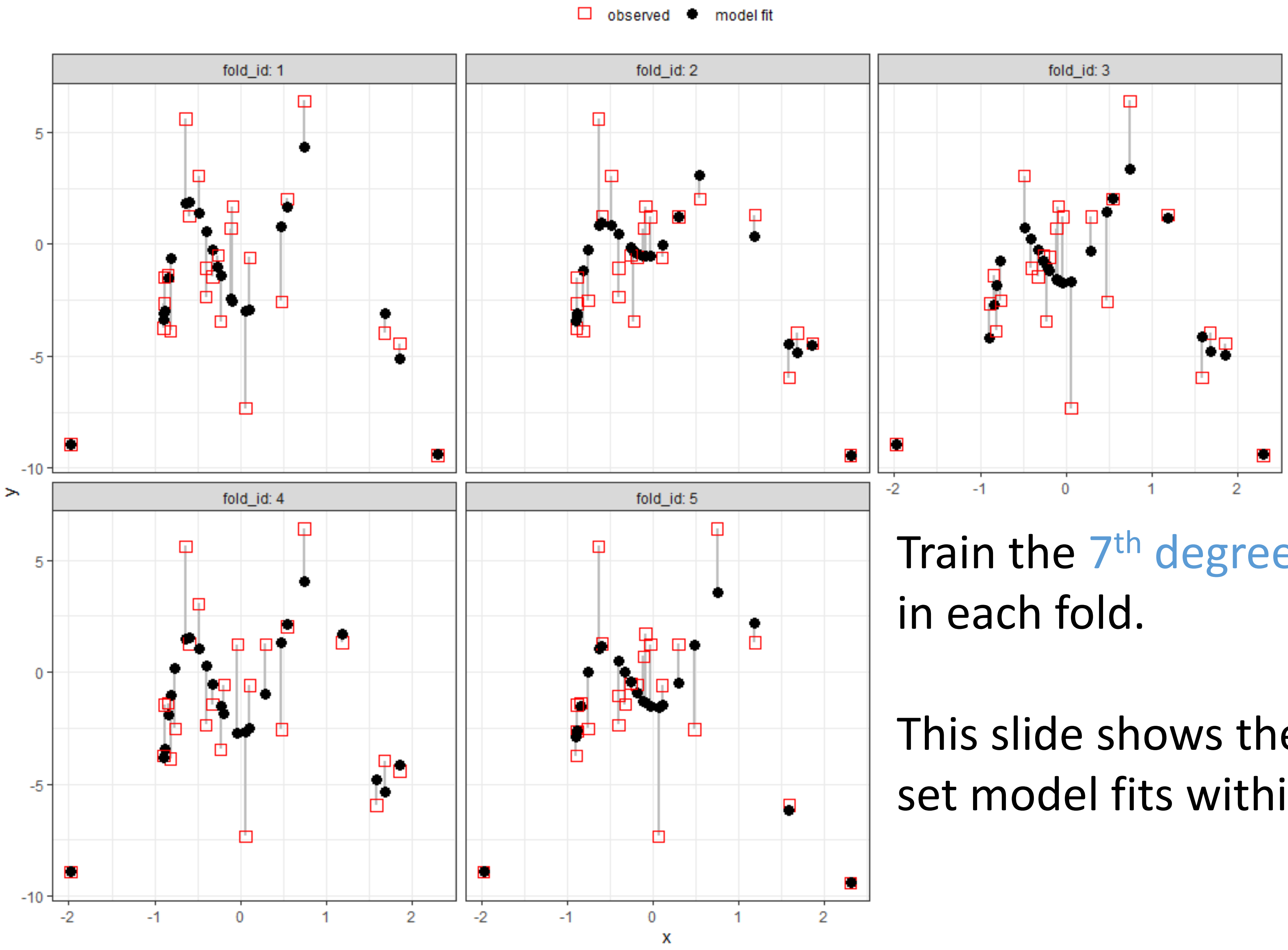
Predict the "fine" visualization grid we created earlier based on the trained model in each fold.

This is **RARELY** done…but helps demonstrate what the model "thinks" is happening…



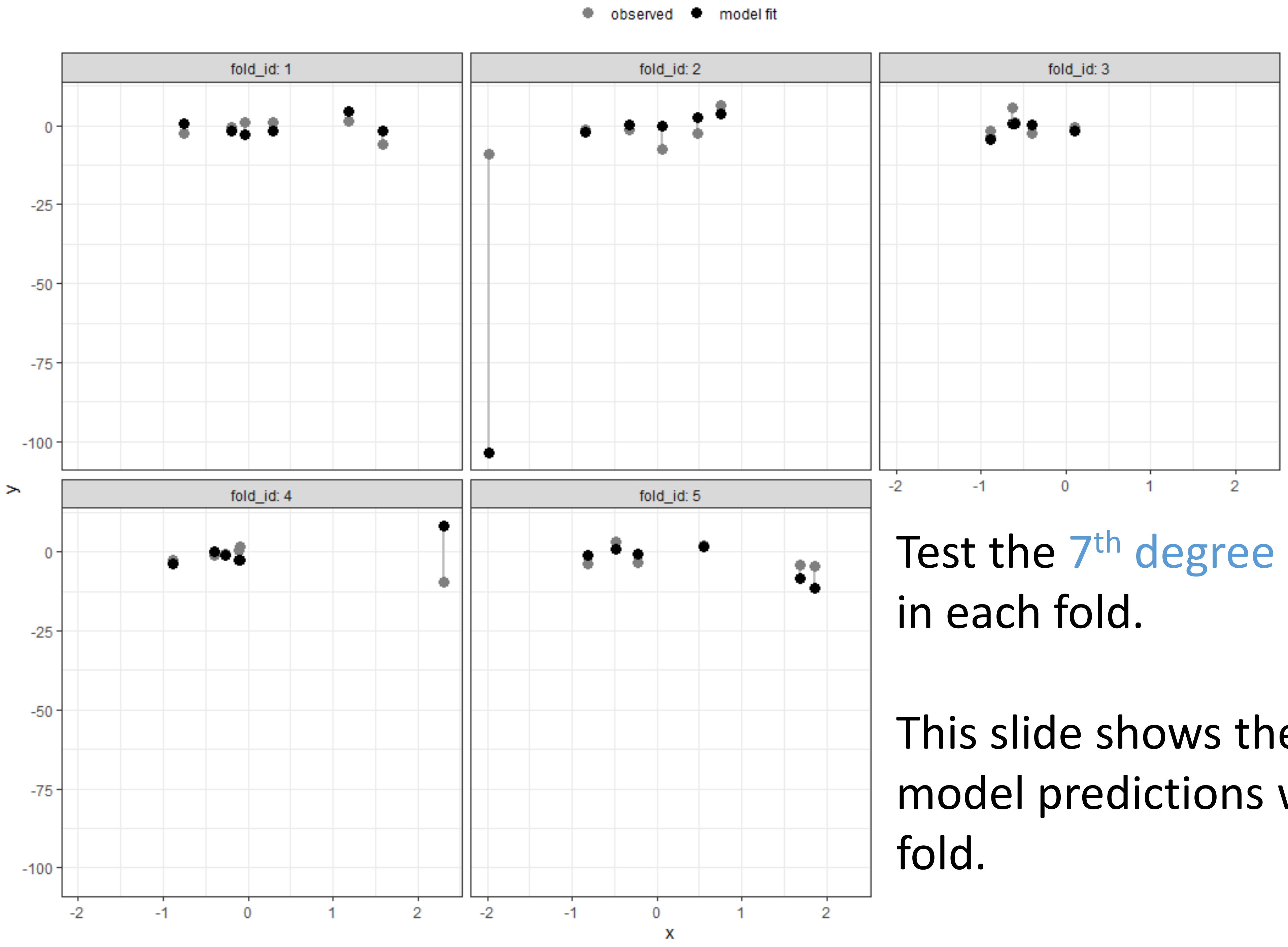
The model predicted trend is **consistent** across all folds!

Insensitive to the training data!



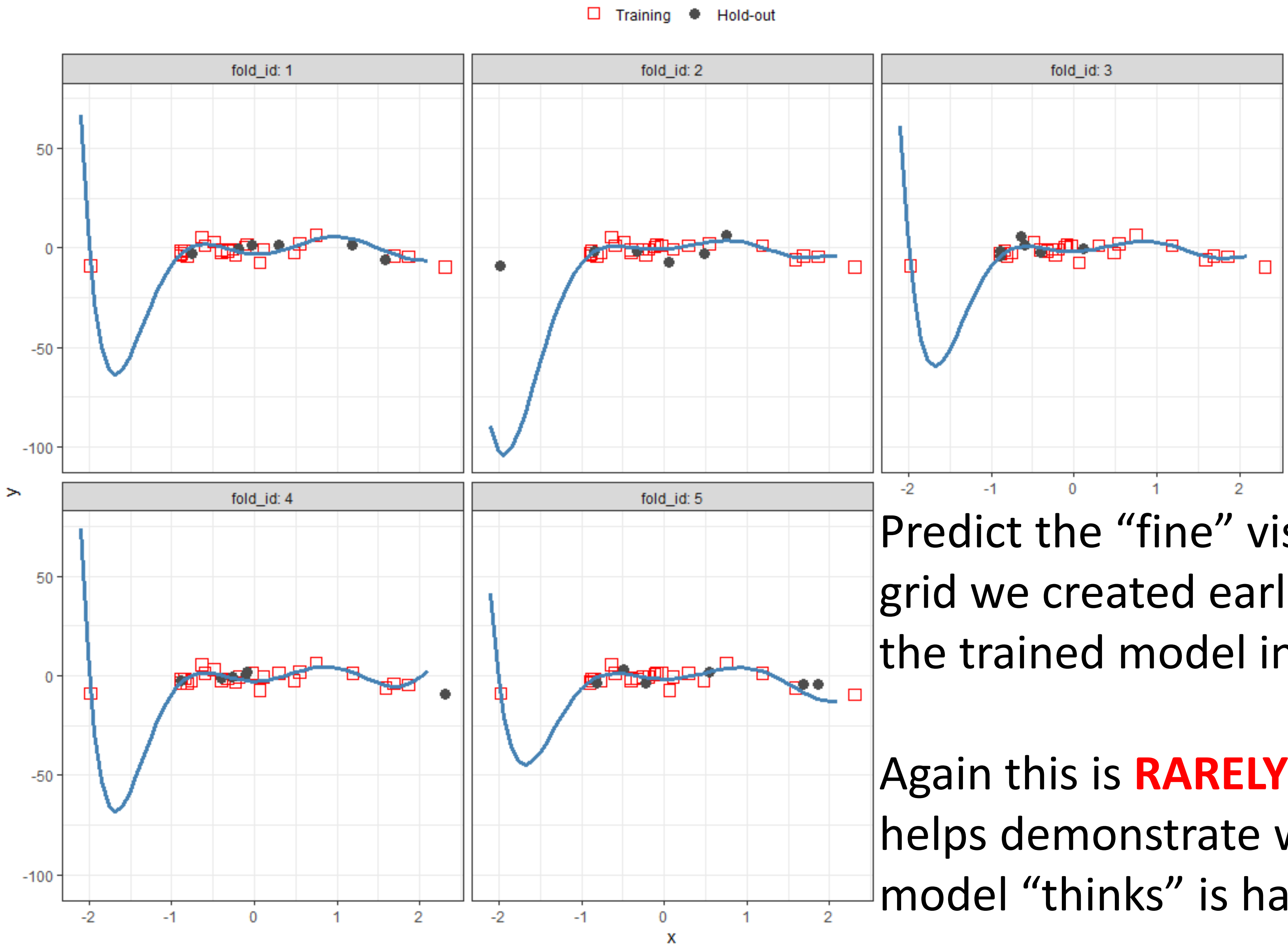
Train the 7th degree polynomial in each fold.

This slide shows the training set model fits within each fold.



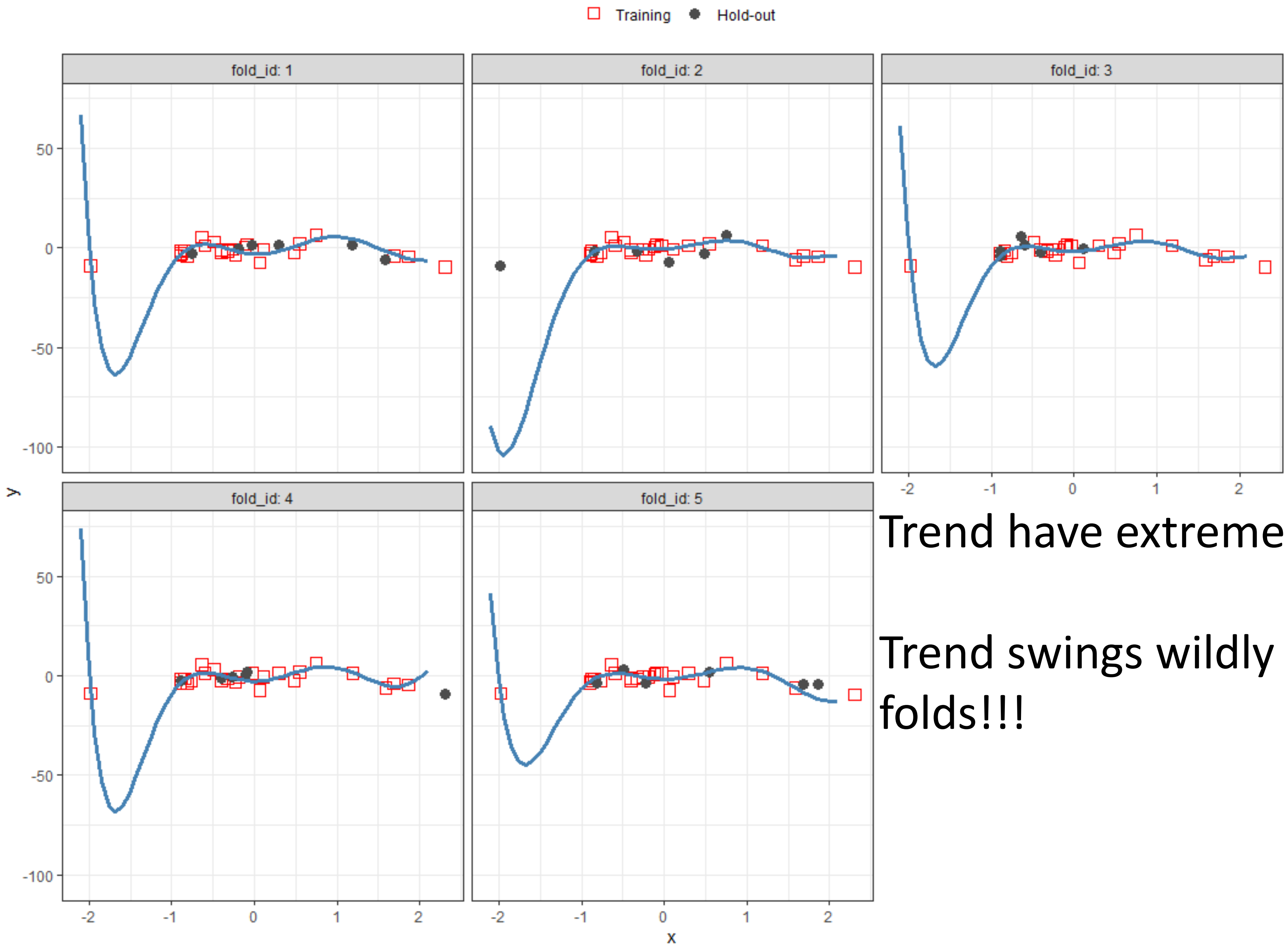
Test the 7th degree polynomial in each fold.

This slide shows the test set model predictions within each fold.



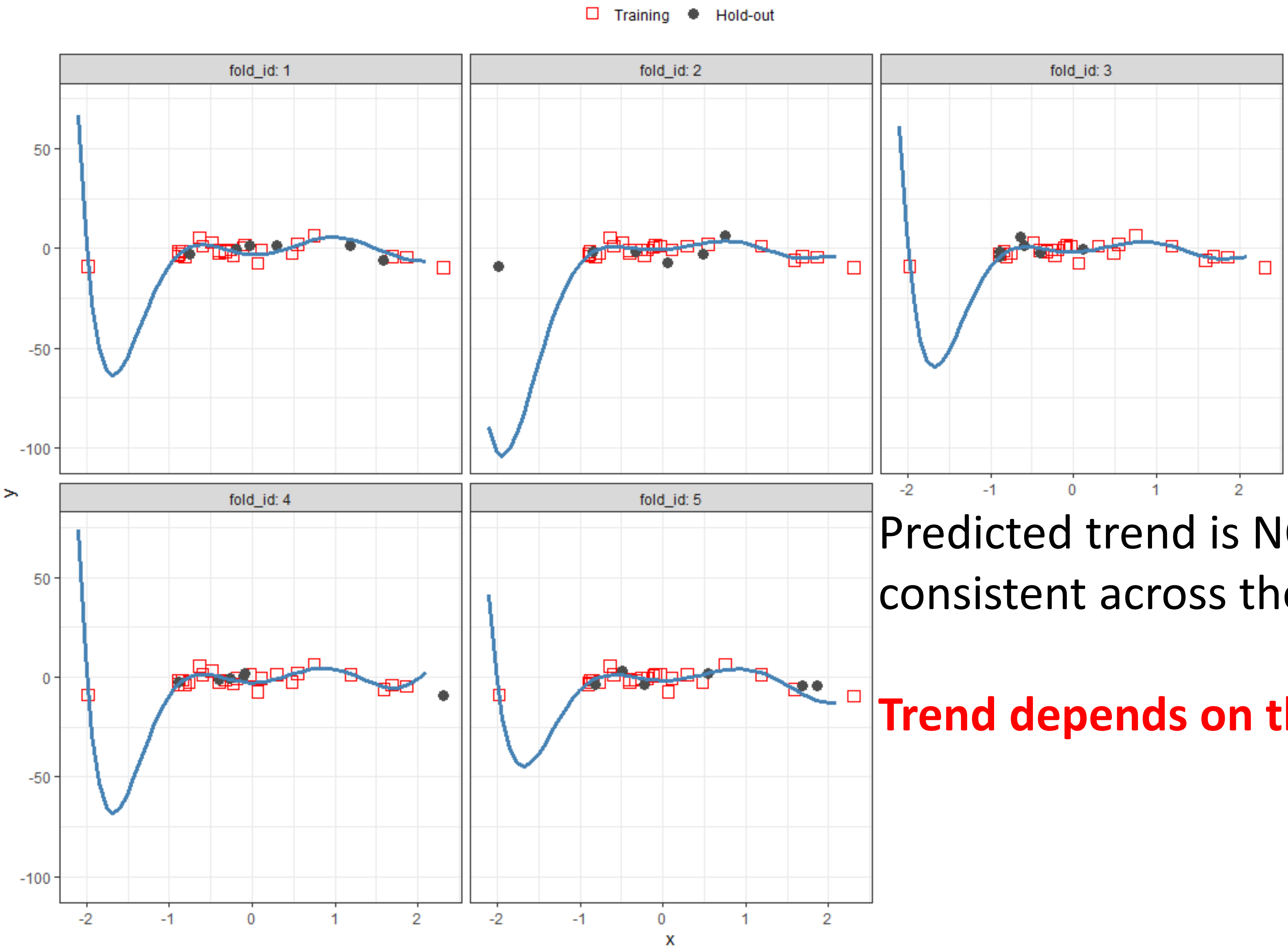
Predict the "fine" visualization grid we created earlier based on the trained model in each fold.

Again this is **RARELY** done…but helps demonstrate what the model "thinks" is happening…



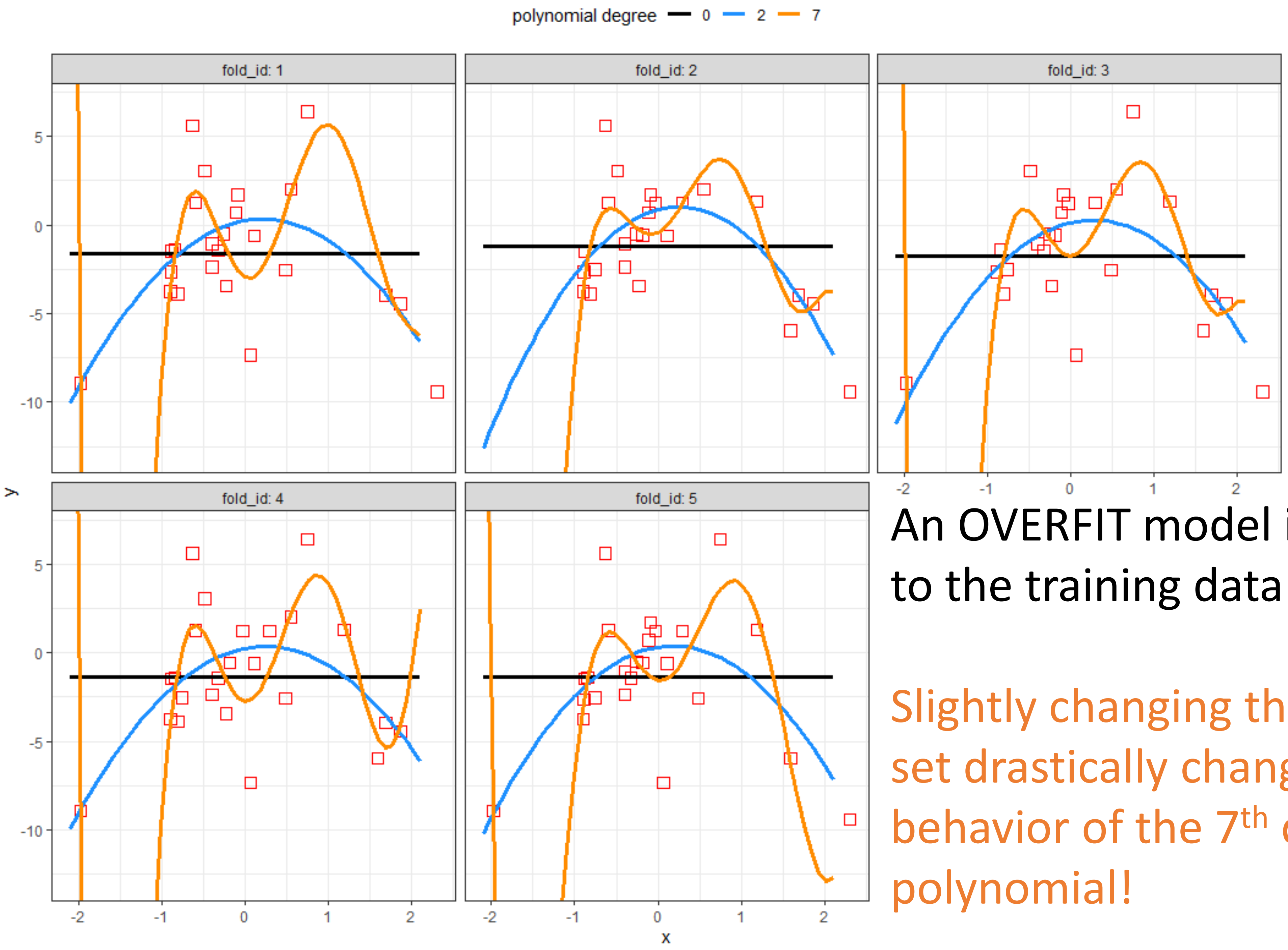
Trend have extreme behavior!!

Trend swings wildly across the folds!!!



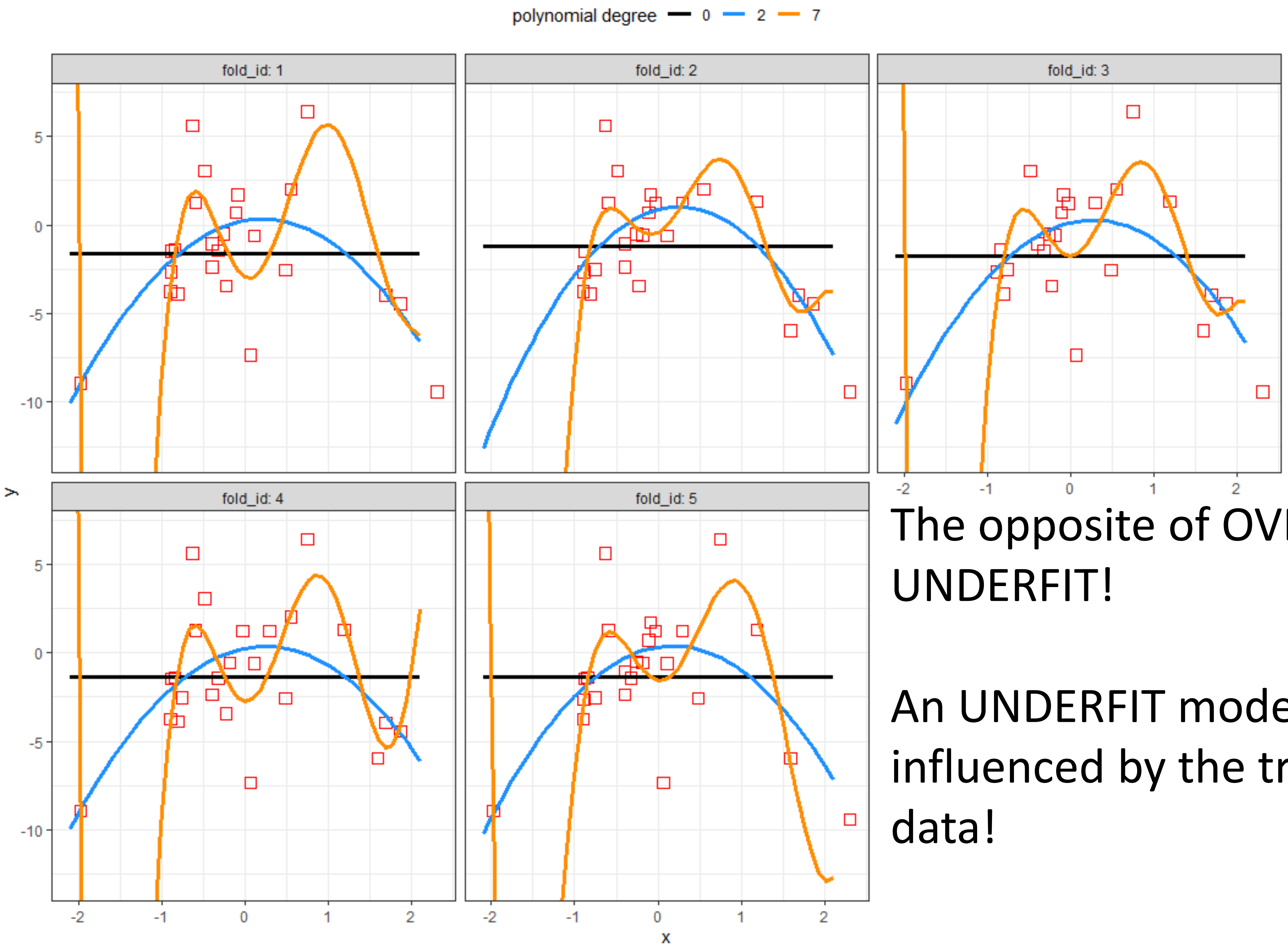
Predicted trend is NOT consistent across the folds!

Trend depends on the data!



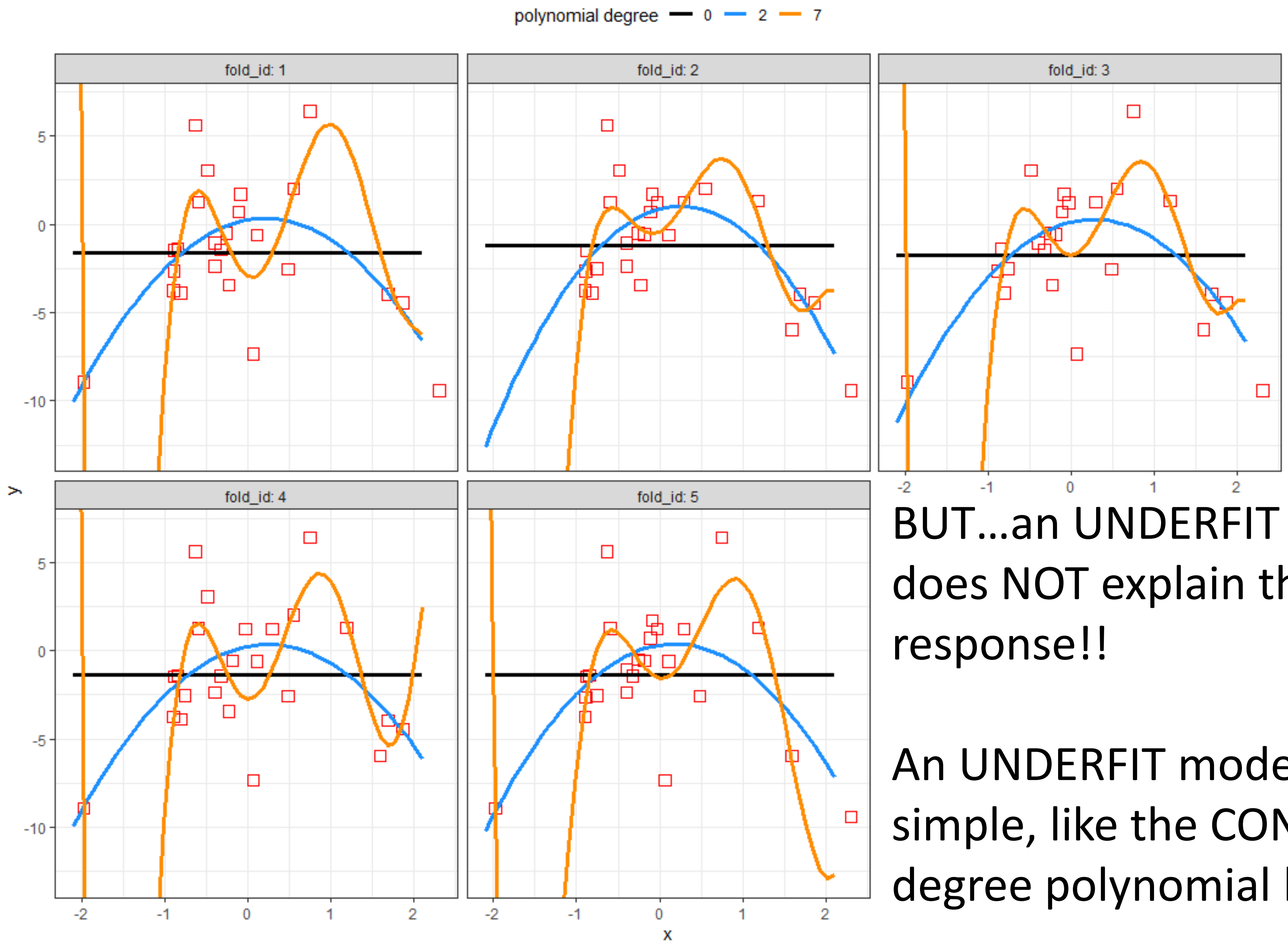
An OVERFIT model is sensitive to the training data!

Slightly changing the training set drastically changes the behavior of the 7th degree polynomial!



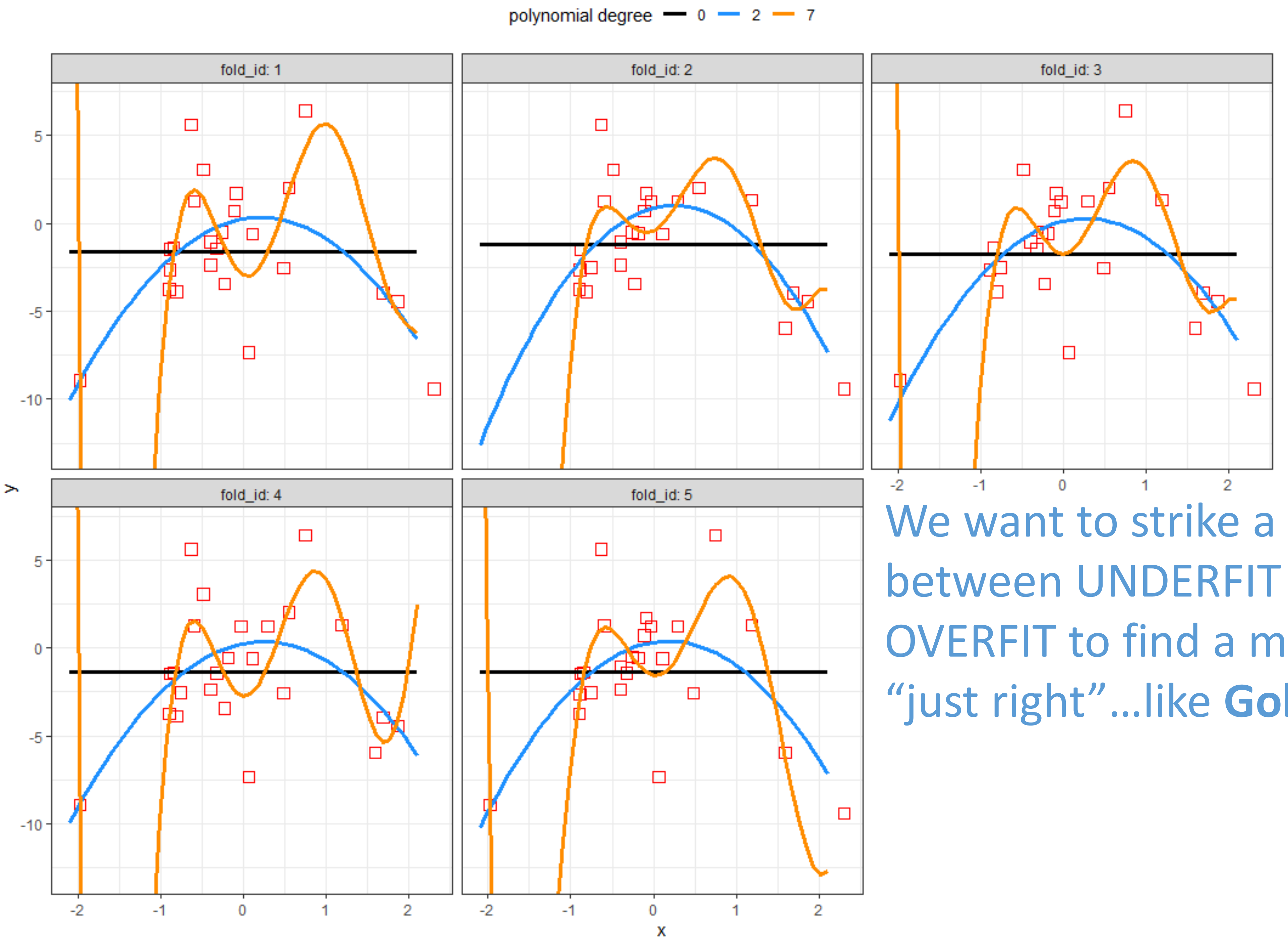
The opposite of OVERFIT is UNDERFIT!

An UNDERFIT model is NOT influenced by the training data!



BUT…an UNDERFIT model does NOT explain the response!!

An UNDERFIT model is TOO simple, like the CONSTANT (0th degree polynomial here).

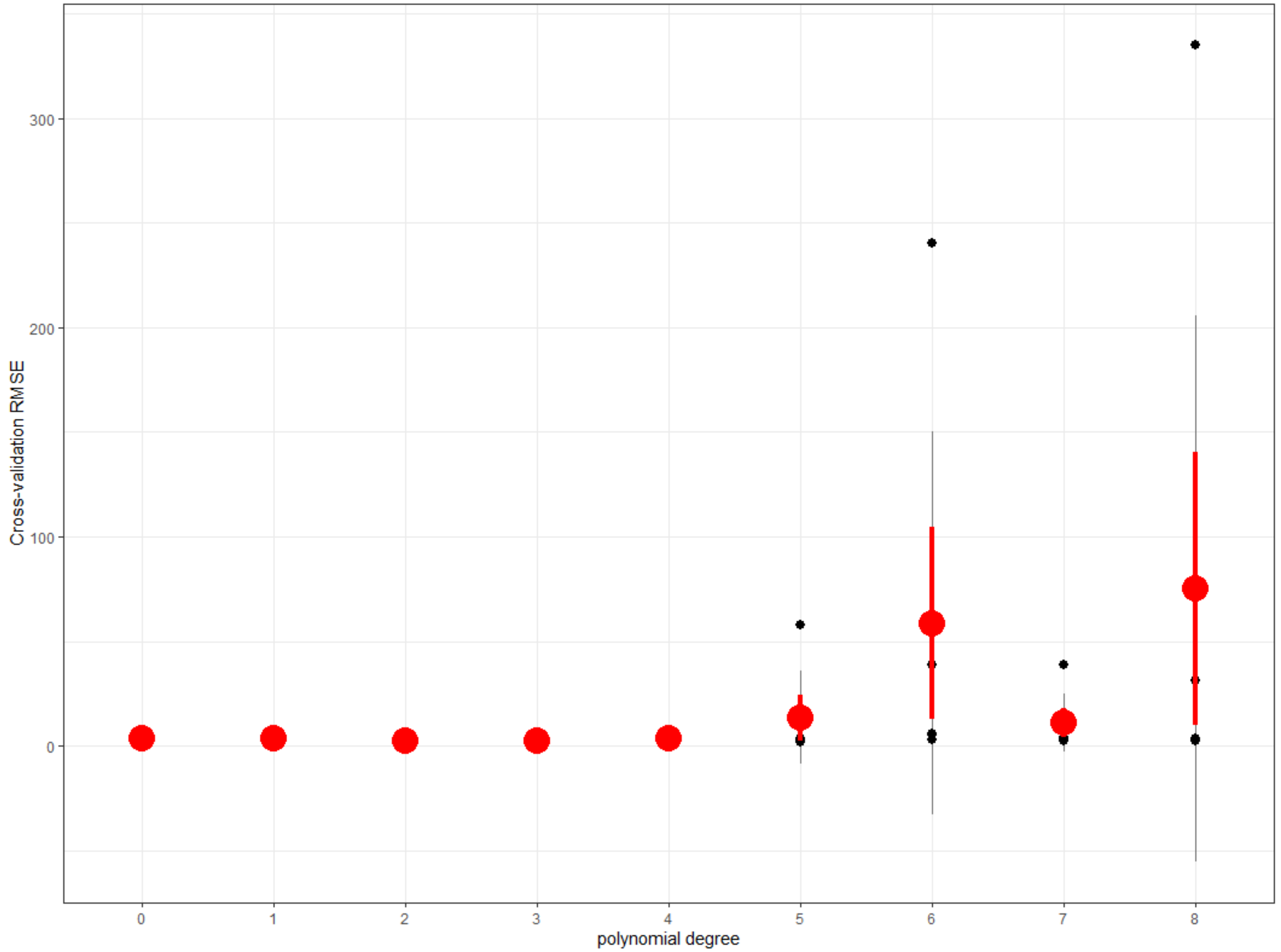


We want to strike a balance between UNDERFIT and OVERFIT to find a model that is "just right"…like **Goldilocks**!

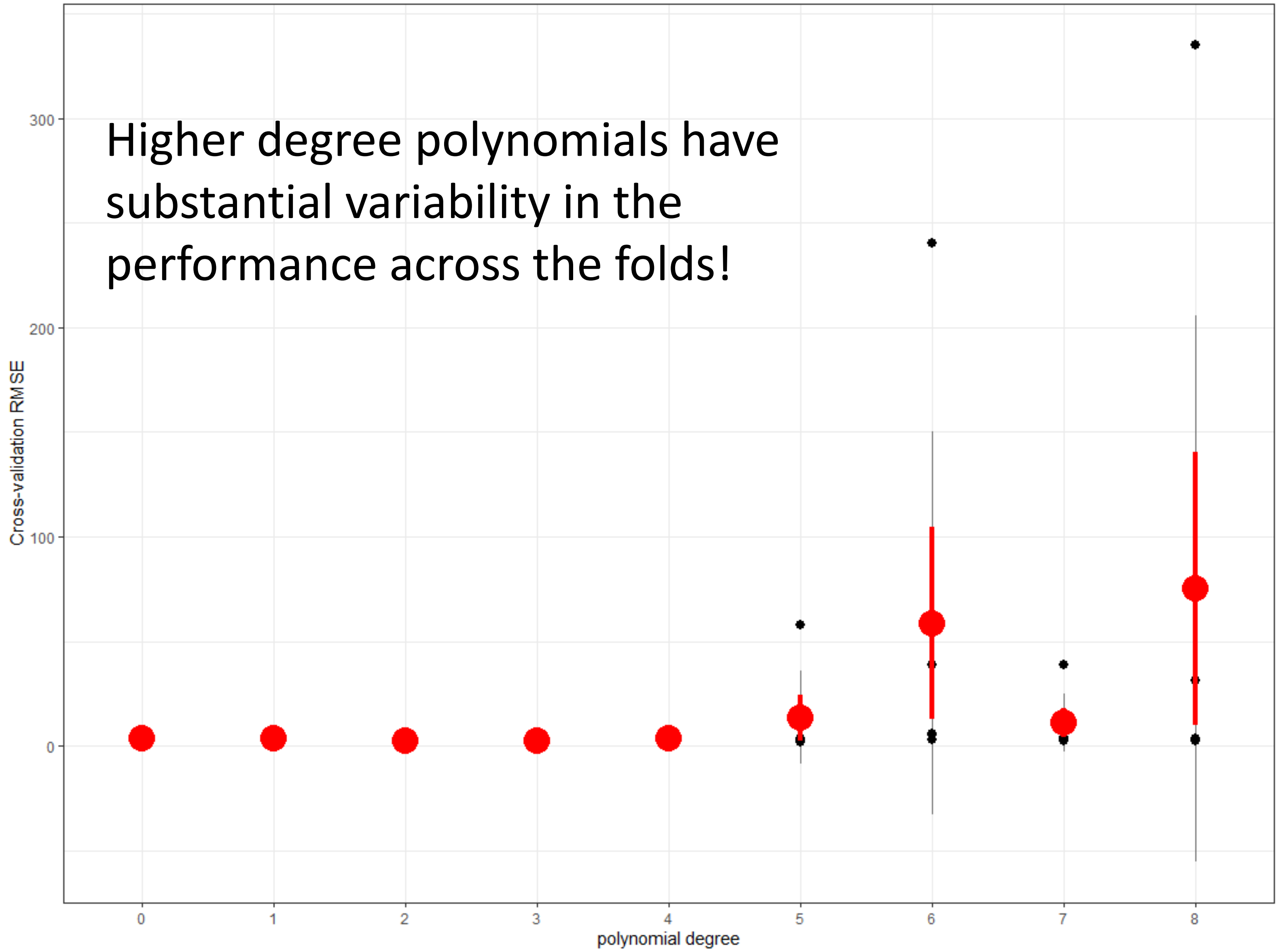
We already calculated the RMSE in each fold for the quadratic relationship and averaged over all folds

- Repeat those steps for every other model!
- Each model will therefore have 5 RMSE values (one for each fold).
- Average over the folds to get the representative performance for each model.

Compare the fold averaged RMSE across the 9 models, intervals represent the STANDARD ERROR

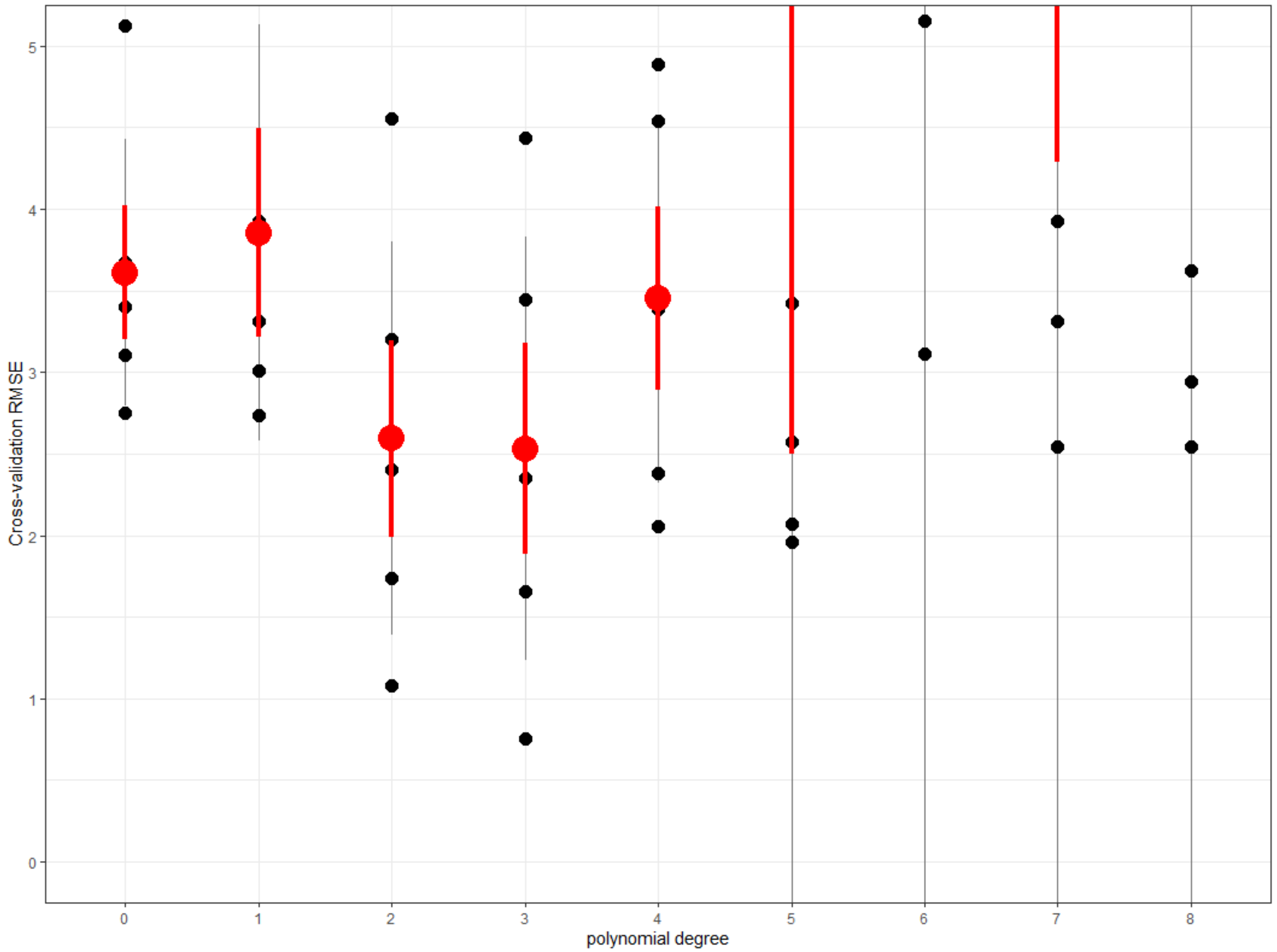


Compare the fold averaged RMSE across the 9 models, intervals represent the STANDARD ERROR

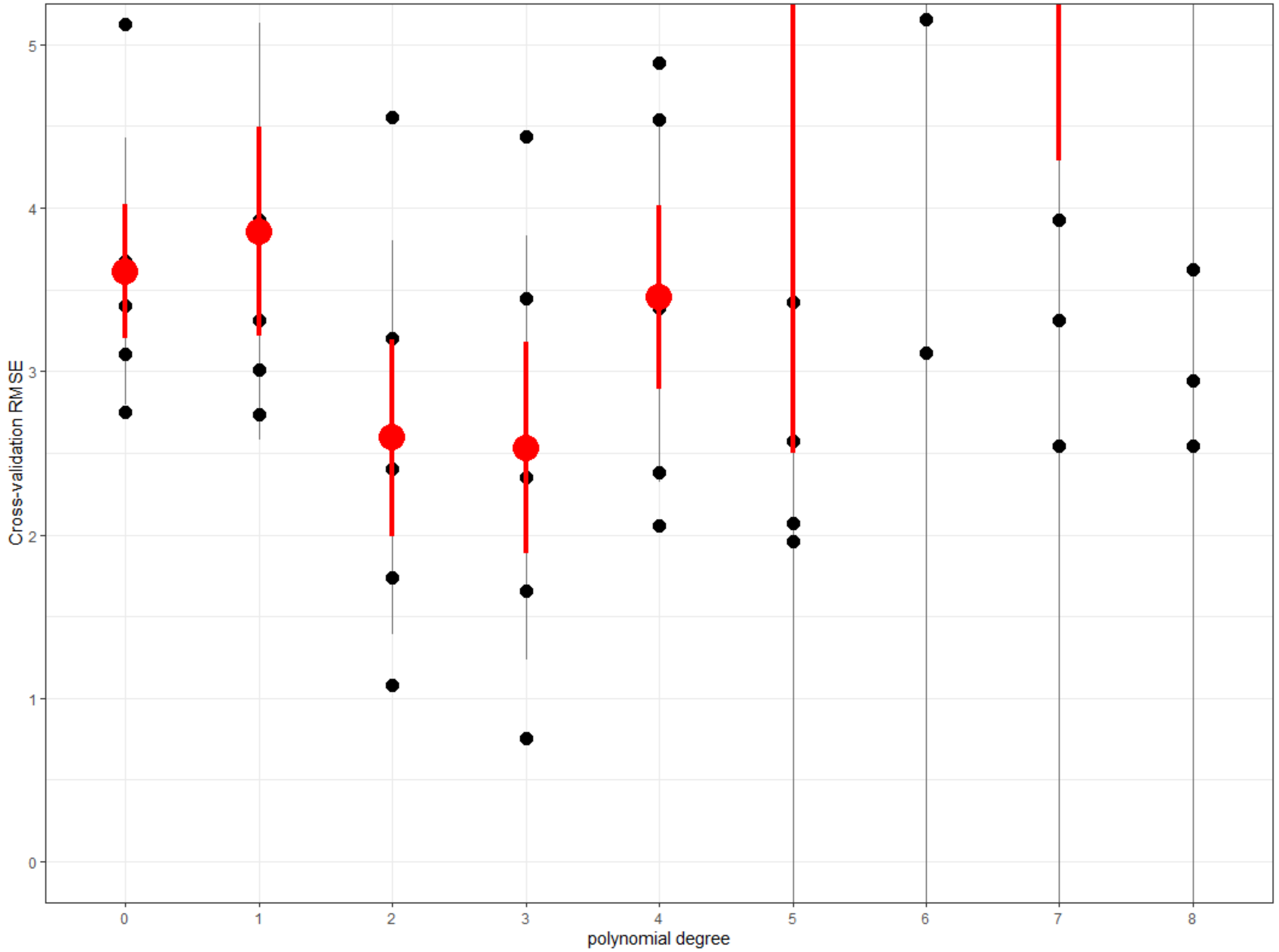


Higher degree polynomials have substantial variability in the performance across the folds!

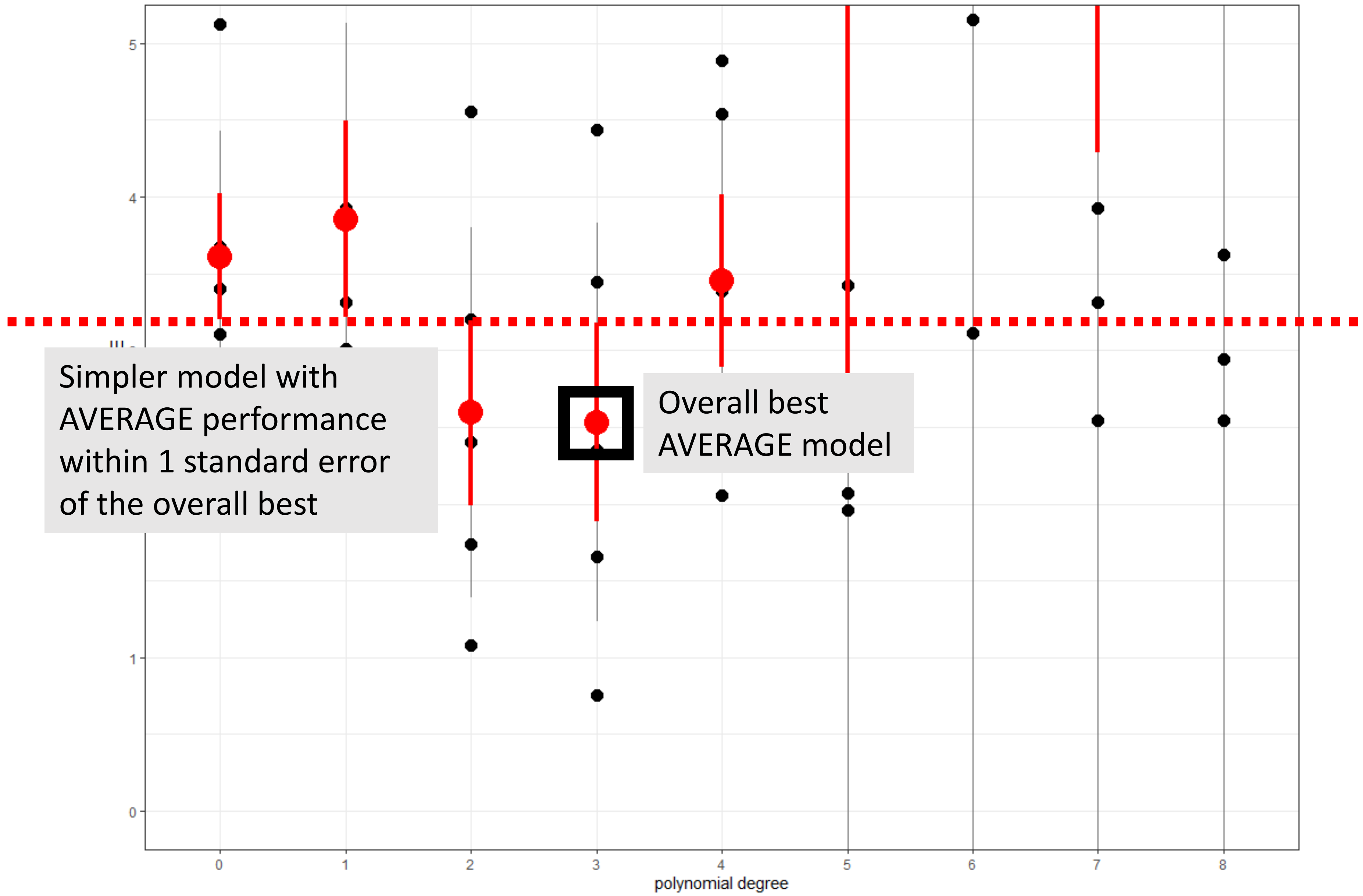
Zoom in to reveal that the quadratic and cubic relationships perform the BEST ON AVERAGE



Look at the STANDARD ERROR interval around the AVERAGE…
The quadratic and cubic models seem to have a lot of "overlap"…



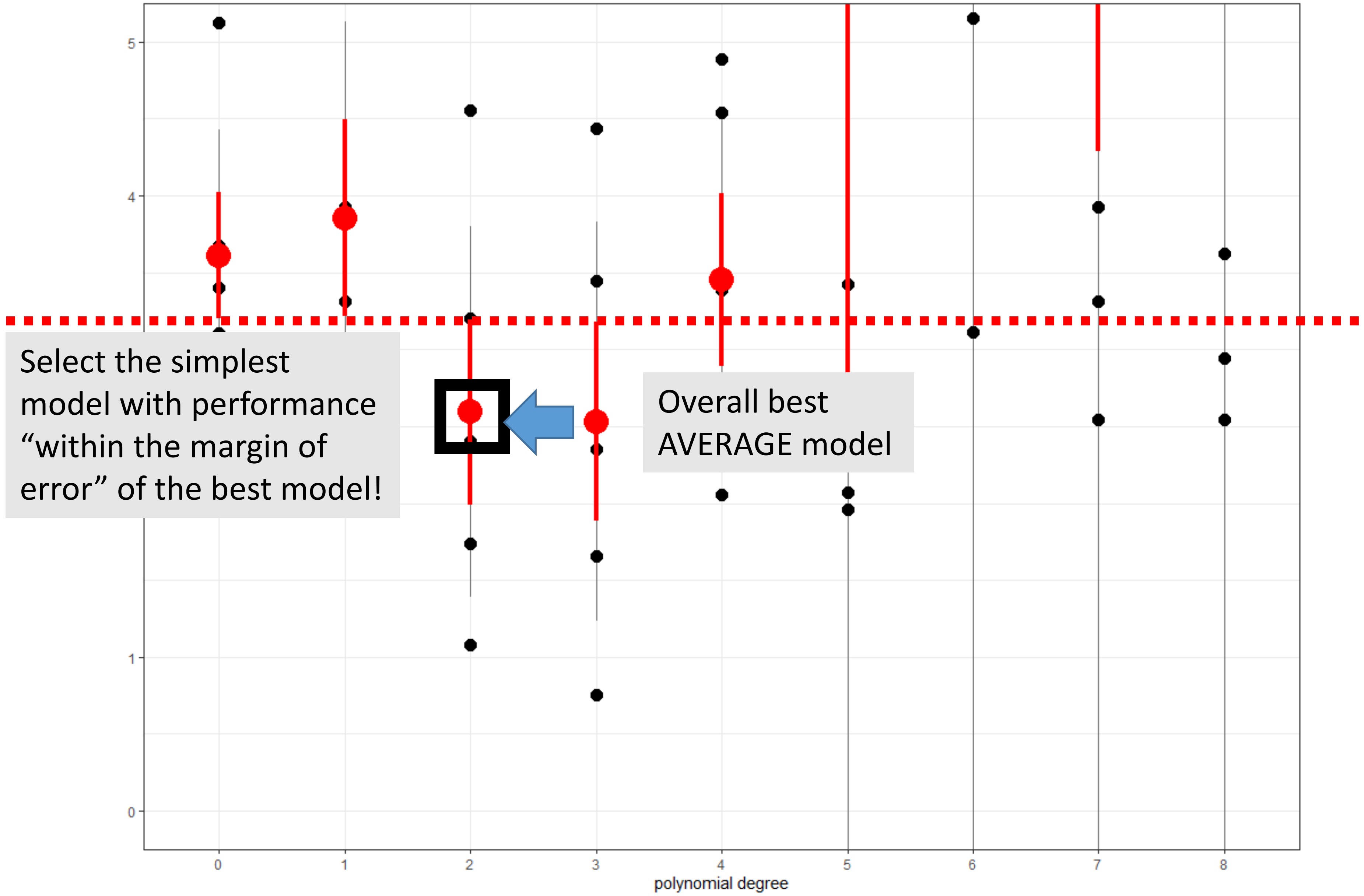
ONE-STANDARD ERROR RULE



Simpler model with AVERAGE performance within 1 standard error of the overall best

Overall best AVERAGE model

ONE-STANDARD ERROR RULE



Select the simplest model with performance “within the margin of error” of the best model!

Overall best AVERAGE model

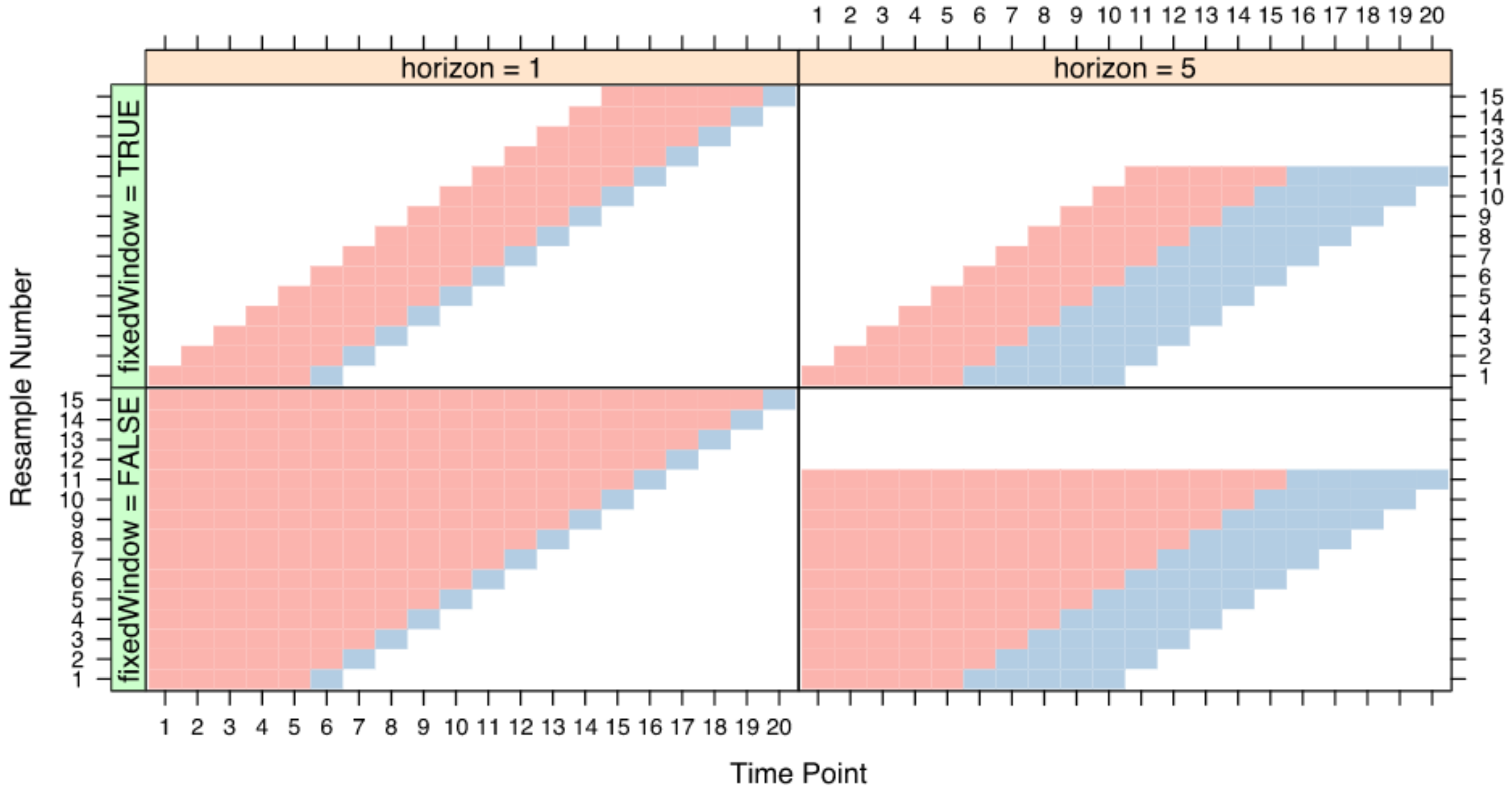
Train, validate, test…

- General guidelines are to first split a data set such that 20% of the data points are only used as a complete hold-out set.
- Perform cross-validation on the remaining 80%.
- However…this is just a rule of thumb…it's not the only way!

Time series cross-validation

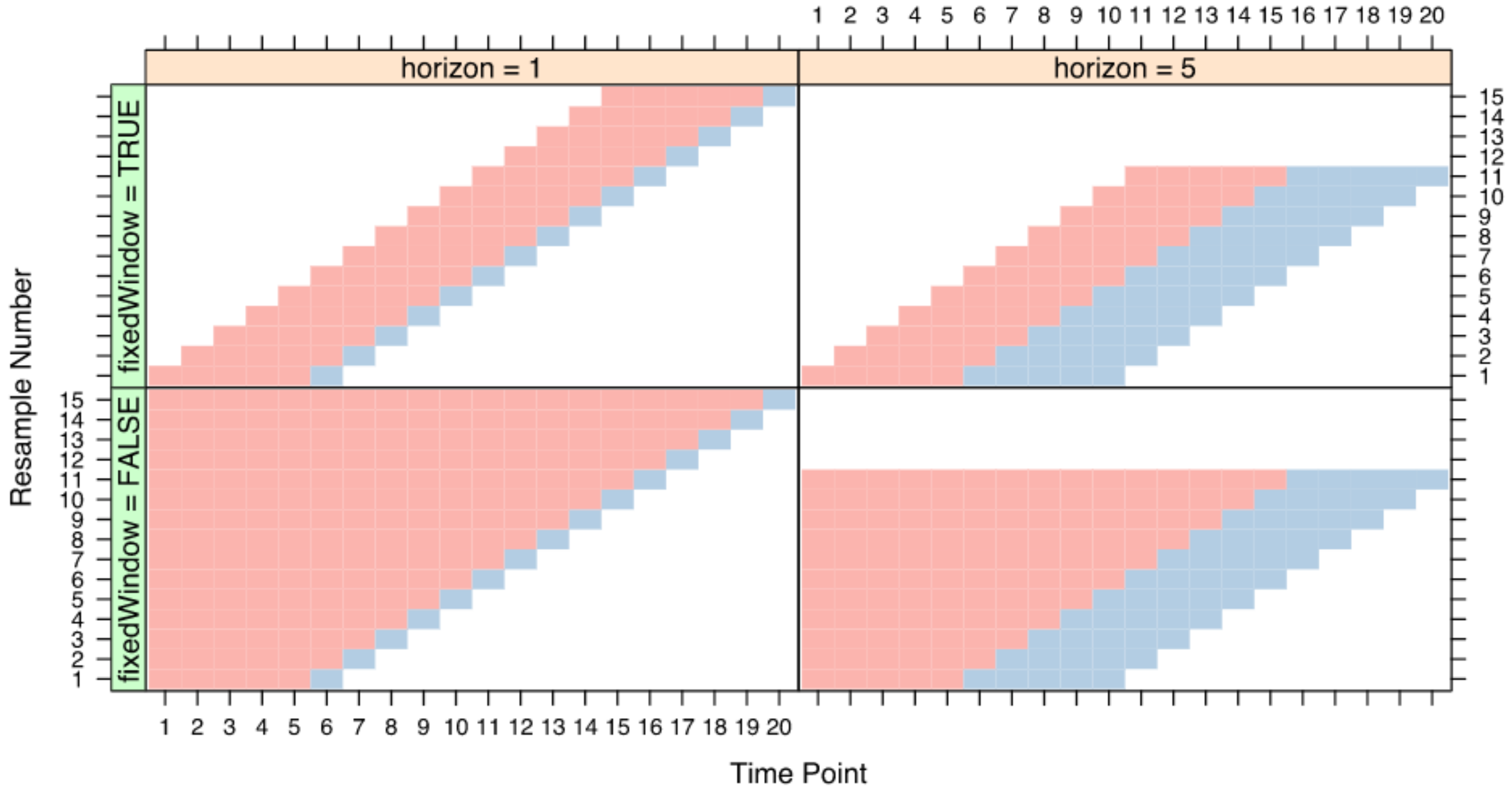
- Time series modeling is a perfect example for when the “conventional” cross-validation approach is inappropriate.
- If we wish to forecast future events, based on previous observations, random partitioning of the data set will break that structure.

Time series windows



From: http://topepo.github.io/caret/data-splitting.html#data-splitting-for-time-series

We will not cover time series forecasting in this course…but remember to think about the structure of your data!!



From: http://topepo.github.io/caret/data-splitting.html#data-splitting-for-time-series

You might have noticed…there are a lot of little steps you have to execute to use cross-validation!

- You will be applying cross-validation to identify the best model in an example…very similar to this one…in homework 02!

- However, you can use the `caret` package to handle all the book-keeping for you!

The code below performs 5-fold cross-validation on the linear through cubic relationships

```
library(caret)

my_ctrl <- trainControl(method = "cv", number = 5,
                        savePredictions = TRUE)

my_metric <- "RMSE"

### train the models with caret

set.seed(71231)
fit_lm_01 <- train(y ~ x,
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)

set.seed(71231)
fit_lm_02 <- train(y ~ x + I(x^2),
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)

set.seed(71231)
fit_lm_03 <- train(y ~ x + I(x^2) + I(x^3),
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)
```

The code below performs 5-fold cross-validation on the linear through cubic relationships

```r
library(caret)

my_ctrl <- trainControl(method = "cv", number = 5,
                        savePredictions = TRUE)

my_metric <- "RMSE"

### train the models with caret

set.seed(71231)
fit_lm_01 <- train(y ~ x,
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)

set.seed(71231)
fit_lm_02 <- train(y ~ x + I(x^2),
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)

set.seed(71231)
fit_lm_03 <- train(y ~ x + I(x^2) + I(x^3),
                   data = my_train,
                   method = "lm",
                   metric = my_metric,
                   preProcess = c("center", "scale"),
                   trControl = my_ctrl)
```

By default `caret` does not save the fold hold-out set predictions, only the summary performance metrics.

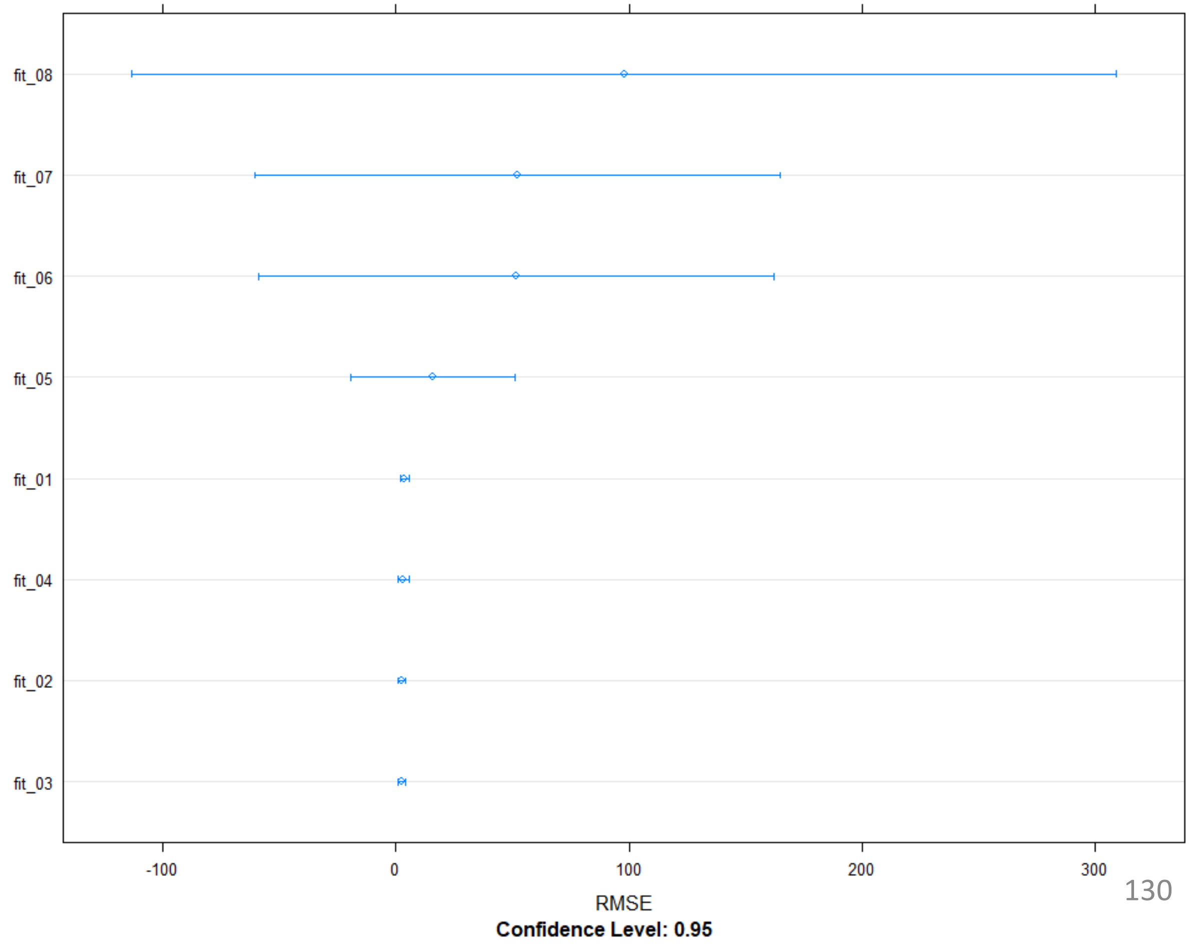
We can tell `caret` to perform pre-processing steps during the cross-validation! We'll talk about this more later, HOML has a nice discussion on pre-processing.

`caret` accepts the formula interface as well.

Homework 02 will show you how to compile the results of the `caret` training

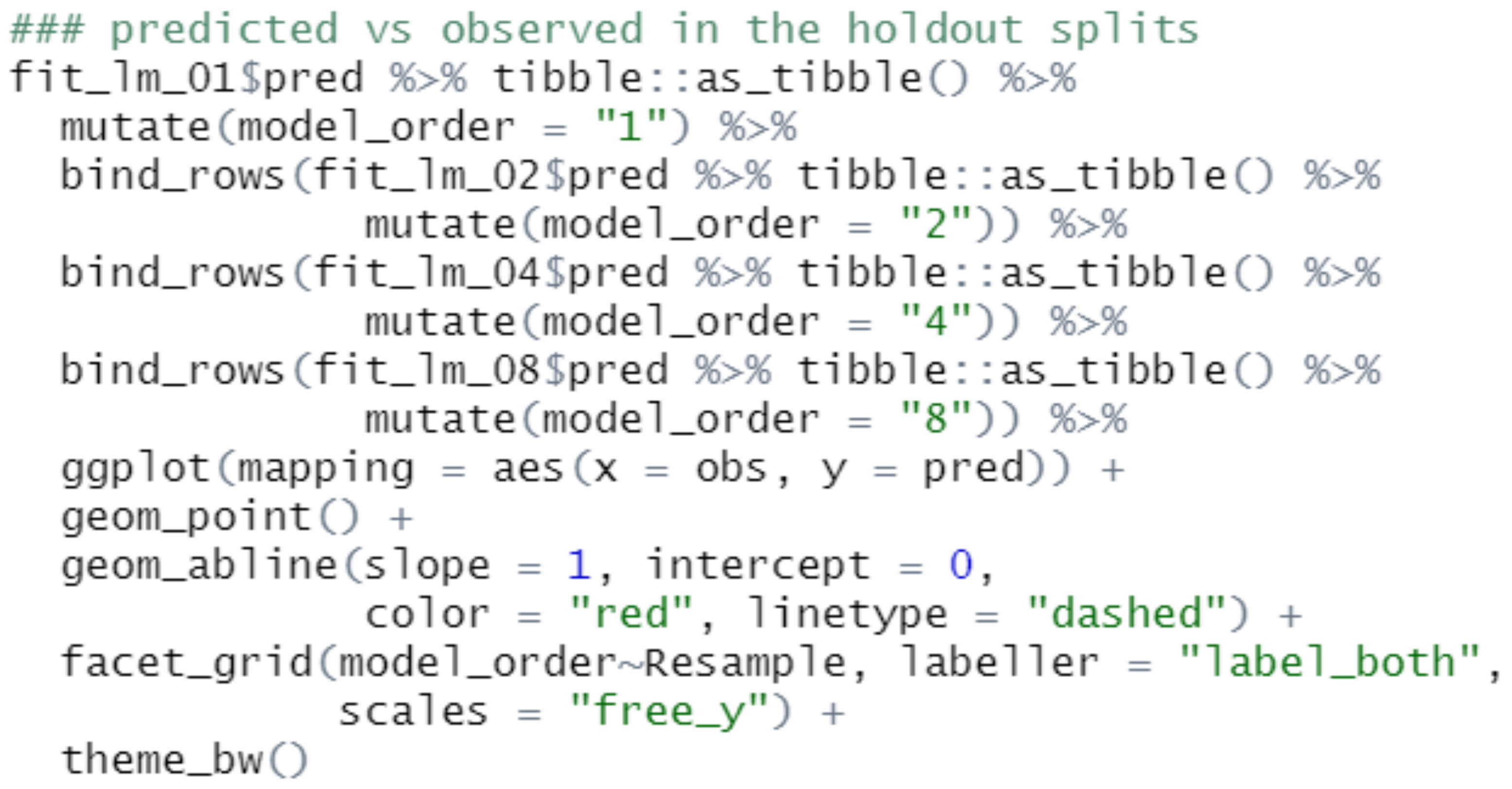
- After training the linear through 8th degree polynomial, can use a default plot method to visualize the results

```
### compare models
caret_results <- resamples(list(fit_01 = fit_lm_01,
                                fit_02 = fit_lm_02,
                                fit_03 = fit_lm_03,
                                fit_04 = fit_lm_04,
                                fit_05 = fit_lm_05,
                                fit_06 = fit_lm_06,
                                fit_07 = fit_lm_07,
                                fit_08 = fit_lm_08))

dotplot(caret_results, metric = "RMSE")
```

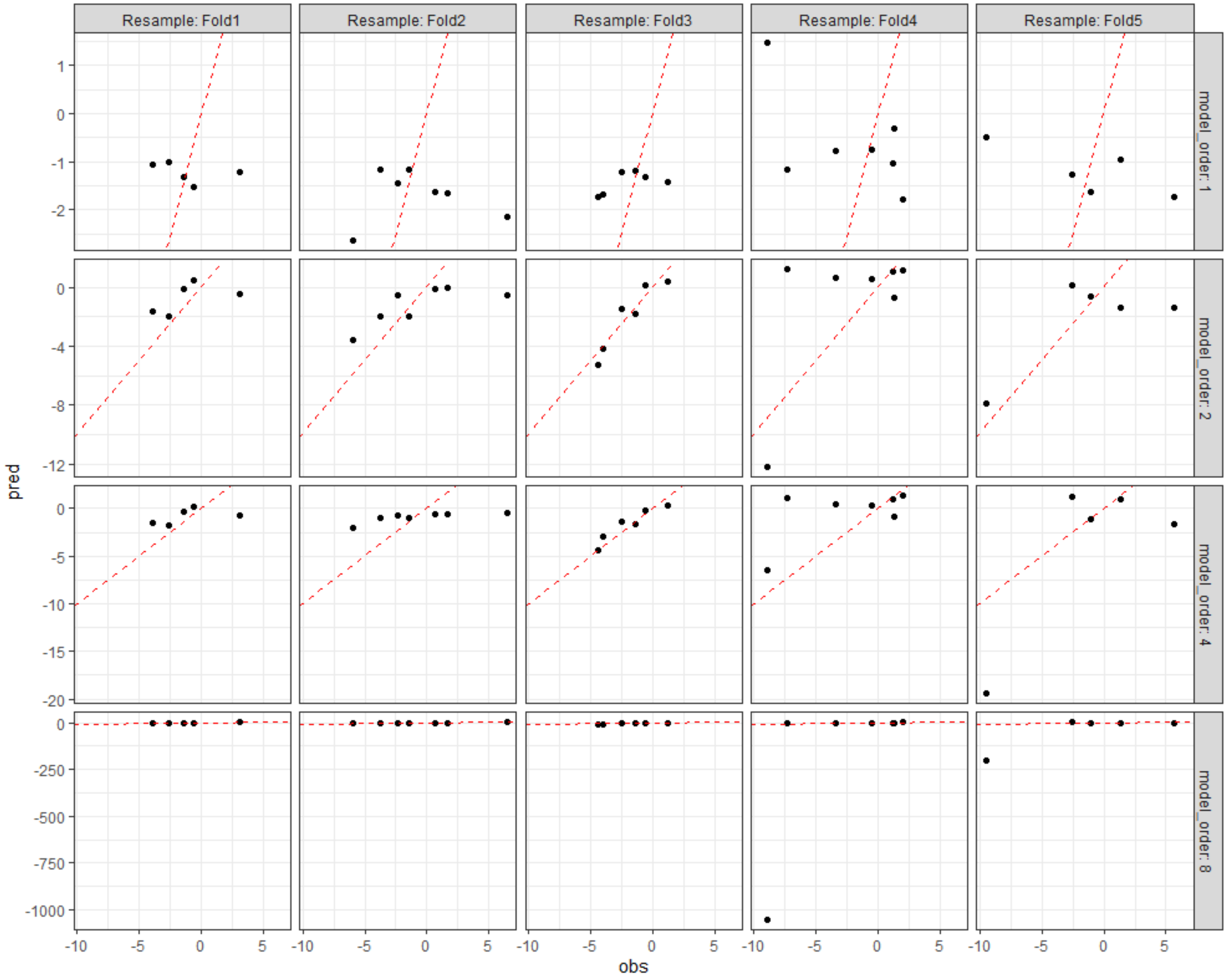


Since we saved the fold hold-out set predictions we can look at a predicted-vs-observed figure across several models

- Code below uses ggplot2 to create the figure for the linear, quadratic, 4th degree, and 8th degree polynomials



```r
### predicted vs observed in the holdout splits
fit_lm_01$pred %>% tibble::as_tibble() %>%
  mutate(model_order = "1") %>%
  bind_rows(fit_lm_02$pred %>% tibble::as_tibble() %>%
              mutate(model_order = "2")) %>%
  bind_rows(fit_lm_04$pred %>% tibble::as_tibble() %>%
              mutate(model_order = "4")) %>%
  bind_rows(fit_lm_08$pred %>% tibble::as_tibble() %>%
              mutate(model_order = "8")) %>%
  ggplot(mapping = aes(x = obs, y = pred)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0,
              color = "red", linetype = "dashed") +
  facet_grid(model_order~Resample, labeller = "label_both",
             scales = "free_y") +
  theme_bw()
```


Resample: Fold1
Resample: Fold2
Resample: Fold3
Resample: Fold4
Resample: Fold5
model_order: 1
model_order: 2
model_order: 4
model_order: 8
pred
obs
1
0
-1
-2
-4
-8
-12
-5
-10
-15
-20
-250
-500
-750
-1000
-10
-5
0
5