Introduction to Modeling

Kylie Ariel Bemis

10/23/2018

Introduction to Modeling

All models are wrong, but some are useful.

– George Box

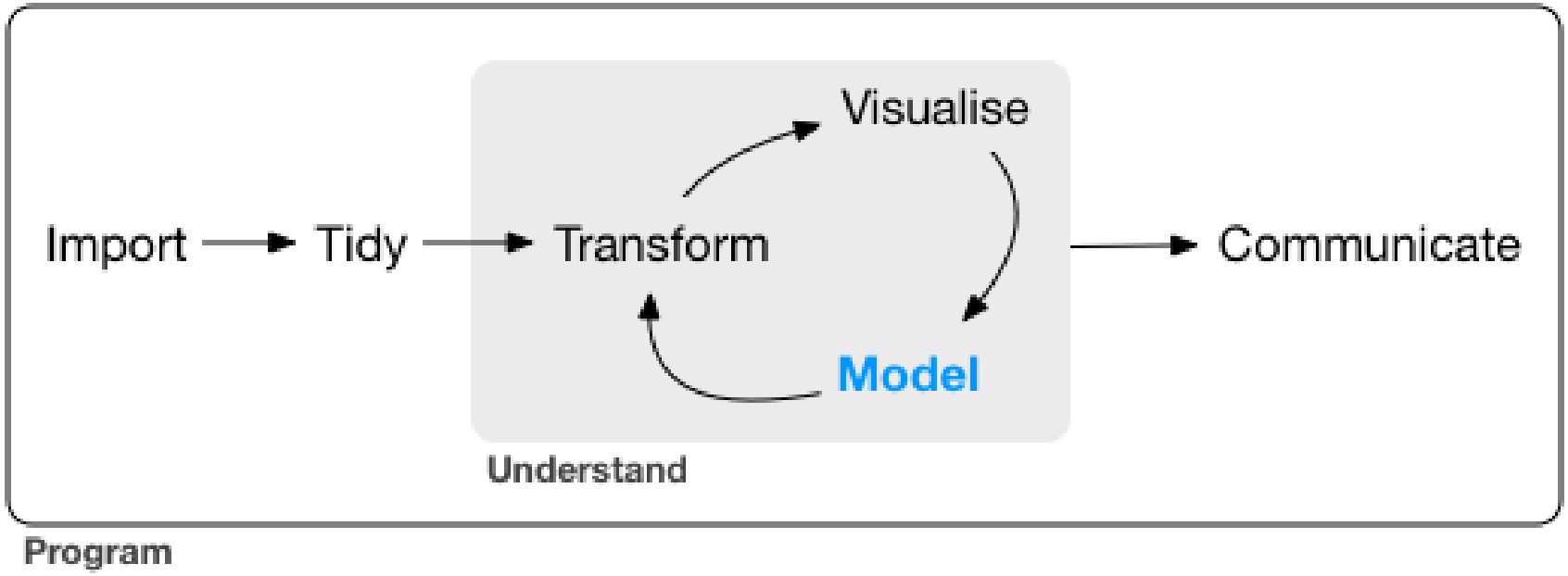


Figure 1: Wickham and Grolemund, *R for Data Science*

Introduction to Modeling

Why do we want to model data?

Models can be used to:

- Predict new values
- Uncover hidden structure
- Perform statistical inference

We will primarily discuss fitting *parametric models*.

- Create a low-dimensional summary of the dataset
- Capture as much of the variation as possible in this summary
- Constrain the ways to do this summary by defining model families

Parametric models

A **family of models** defines the assumptions that form the model:

$$y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n + error$$

$$error \sim N(0, \sigma^2)$$

A **fitted model** finds the model from the family that best fits the data by estimating the parameters, e.g.:

$$y = 1.1 + 2.4 x_1 + ... + 3.2 x_n + error$$

$$error \sim N(0, 1.4)$$

Fitting the model is typically done by optimizing a function of the parameters given the data, such as maximizing a log-likelihood or minimizing the predictive error.

Parametric models in R

Parameter models are described in R using the `formula`.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + error$$

becomes

```
y ~ x1 + x2
```

```
## y ~ x1 + x2
```

Parametric models in R (cont'd)

Transformations and interactions are allowed in the `formula`.

$$log(y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + error$$

becomes

```
log(y) ~ x1 * x2
```

```
## log(y) ~ x1 * x2
```

which expands to

```
log(y) ~ x1 + x2 + x1:x2
```

```
## log(y) ~ x1 + x2 + x1:x2
```

Parametric models in R (cont'd)

Use - to remove terms in the `formula` such as the assumed intercept.

$$y = \beta_1 x_1 + \textit{error}$$

becomes

```
y ~ x1 - 1
```

```
## y ~ x1 - 1
```

Parametric models in R (cont'd)

Use `I()` to disambiguate inline operations like + and *

$$y = \beta_0 + \beta_1(x_1 + x_2) + error$$

becomes

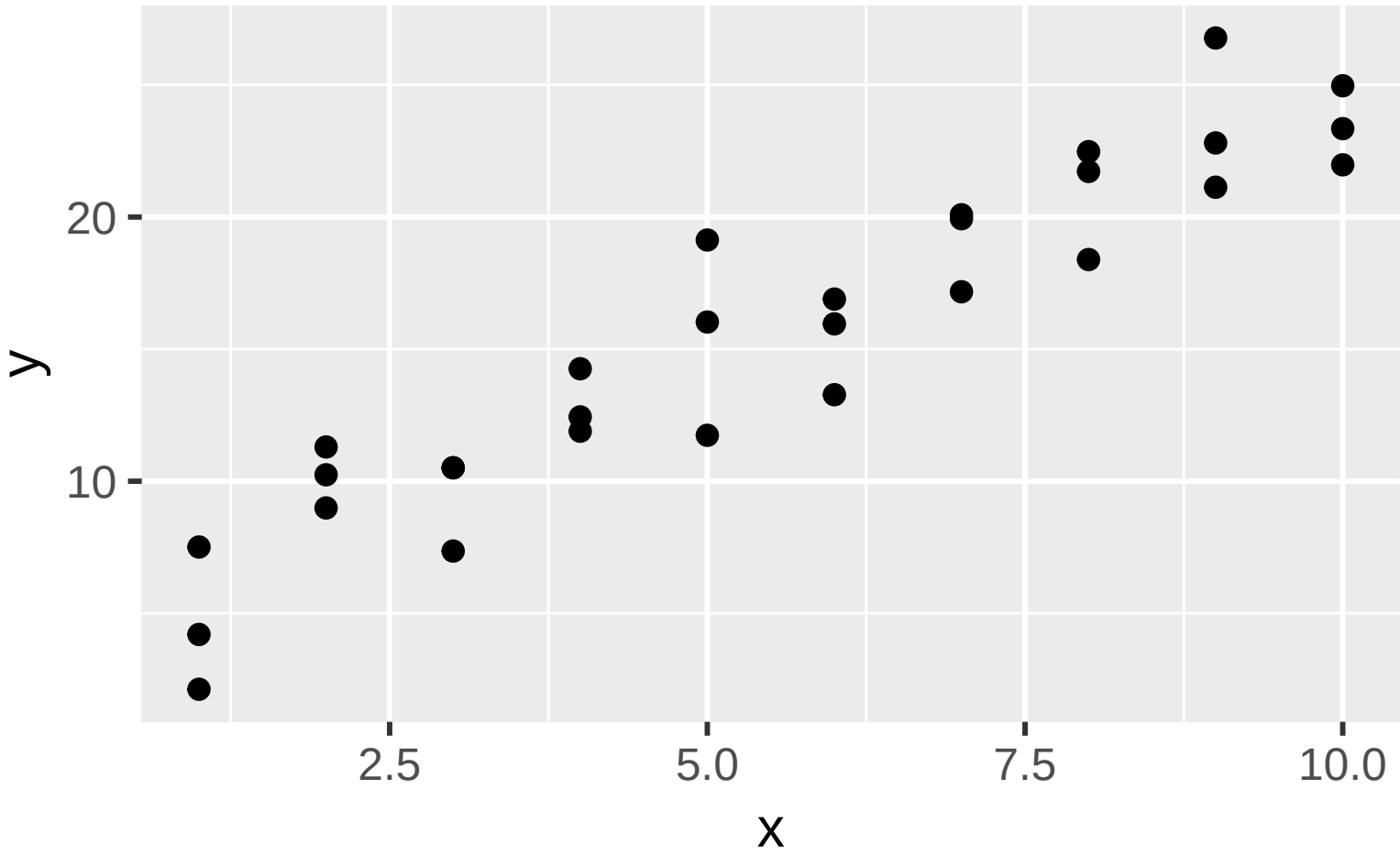
```
y ~ I(x1 + x2)
```

```
## y ~ I(x1 + x2)
```

Simulated data to model

Consider the following simulated data:

```
library(modelr)
ggplot(sim1, aes(x=x, y=y)) + geom_point()
```



Fitting a simple model in R

Fit a simple linear model to it:

```
fit1 <- lm(y ~ x, data=sim1)
fit1
```

```
##
## Call:
## lm(formula = y ~ x, data = sim1)
##
## Coefficients:
## (Intercept)            x
##       4.221        2.052
```

- `x` is the **predictor** or **explanatory** variable
- `y` is the **response** variable

Get model parameters and predicted values

```
coef(fit1) # get parameters estimates
```

```
## (Intercept)           x
##    4.220822    2.051533
```

```
fitted(fit1) # get predicted values
```

```
##         1         2         3         4         5         6
##  6.272355  6.272355  6.272355  8.323888  8.323888  8.323888 10.37542
##         8         9        10        11        12        13         1
## 10.375421 10.375421 12.426954 12.426954 12.426954 14.478487 14.47848
##        15        16        17        18        19        20         2
## 14.478487 16.530020 16.530020 16.530020 18.581553 18.581553 18.58155
##        22        23        24        25        26        27         2
## 20.633087 20.633087 20.633087 22.684620 22.684620 22.684620 24.73615
##        29        30
## 24.736153 24.736153
```

Get residuals (error terms) and number of observations

```
resid(fit1) # get residuals (error terms)
```

```
##           1           2           3           4           5
## -2.072442018  1.238279125 -4.146882207  0.664969362  1.919217378
##           6           7           8           9          10
##  2.972935148 -3.019056466  0.129928252  0.136179642  0.007634878
##          11          12          13          14          15
## -0.534352991  1.831009860  4.651562487 -2.740466108  1.546366596
##          16          17          18          19          20
## -3.256043368 -0.574045413  0.364775796  1.504439222 -1.409703118
##          21          22          23          24          25
##  1.354755377  1.092815968 -2.242173633  1.842466099  4.092390235
##          26          27          28          29          30
##  0.120490168 -1.556314330  0.231946675 -1.389730610 -2.760952005
```

```
nobs(fit1) # get number of observations
```

```
## [1] 30
```

`modelr` versus `stats` (from base R)

The `stats` package is distributed with all default R installations. It includes functions such as:

- `lm`: fit linear models
- `glm`: fit generalized linear models
- `fitted`: get predicted values
- `resid`: get residuals

The `modelr` package is part of the `tidyverse` (but is not loaded automatically with `library(tidyverse)`), and includes some convenience functions that make it easier to work with models with the %>% operator. A few notable functions include:

- `add_predictions`: adds predicted values to a dataset
- `add_residuals`: adds residuals to a dataset
- `resample_partition`: split data into testing and training sets
- `crossv_kfold`: split data into `k` partitions for cross-validation

Evaluating model quality

Now that we've fit a model, how do we know if it's good or not? There are many ways to evaluate a fitted model.

- Goodness of fit
  - Log-likelihood
  - R-squared
- Information criteria
  - Akaike Information Criterion (AIC)
  - Bayesian Information Criterion (BIC)
- Predictive ability
  - Root-mean-squared error
  - Mean absolute error
  - Sensitivity and specificity
- Visualization

For this course, we will focus on **prediction** and **visualization**.

Predictions

It is often a good idea to plot the predicted values with the original data to visually evaluate whether the model appears to be a good fit for the data or not.

For linear models, predicted values ($\hat{y}_i$) are defined as:

$$\hat{y}_i = \beta_0 + \beta_1 x_{1i} + ... + \beta_n x_{ni}$$

Plot the predictions

Plot the predictions using `modelr`:

```
sim1 %>%
  add_predictions(fit1) %>%
  ggplot(aes(x=x)) +
  geom_point(aes(y=y)) +
  geom_line(aes(y=pred))
```

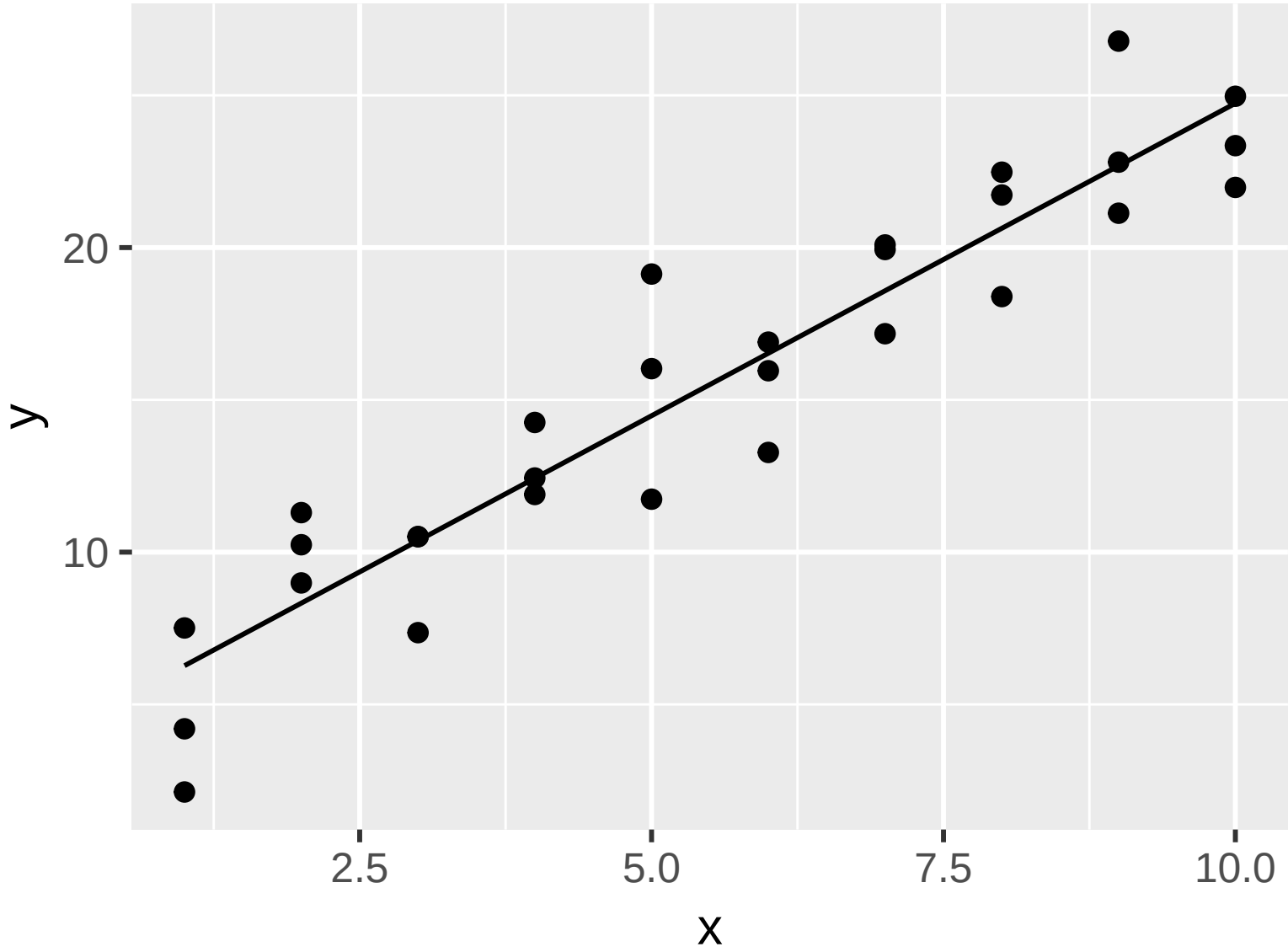
where

`modelr::add_predictions(fit1)`

is equivalent to:

`mutate(pred = fitted(fit1))`

Plot the predictions (cont'd)



Measures of predictive accuracy

Calculate from `fit1` using base R:

```
sqrt(mean(resid(fit1)^2)) # root-mean-square-error
```

```
## [1] 2.128181
```

```
mean(abs(resid(fit1))) # mean absolute error
```

```
## [1] 1.713477
```

Measures of predictive accuracy (cont'd)

Calculate using `modelr`:

```
rmse(fit1, sim1) # root-mean-square-error
```

```
## [1] 2.128181
```

```
mae(fit1, sim1) # mean absolute error
```

```
## [1] 1.713477
```

Residuals

Plotting the residuals (errors) is a good way of evaluating a model visually.

We expect the true error to be randomly distributed (with a normal distribution for linear models), so plotting the residuals against the other variables should show random noise with no systematic pattern.

A systematic pattern when plotted with one of the variables indicates a relationship that your model has not adequately captured.

Residuals (e_i) are defined as:

$$e_i = y_i - \hat{y}_i$$

Plotting residuals

```
sim1 %>%
  add_residuals(fit1) %>%
  ggplot(aes(x=x, y=resid)) +
  geom_point()
```

where

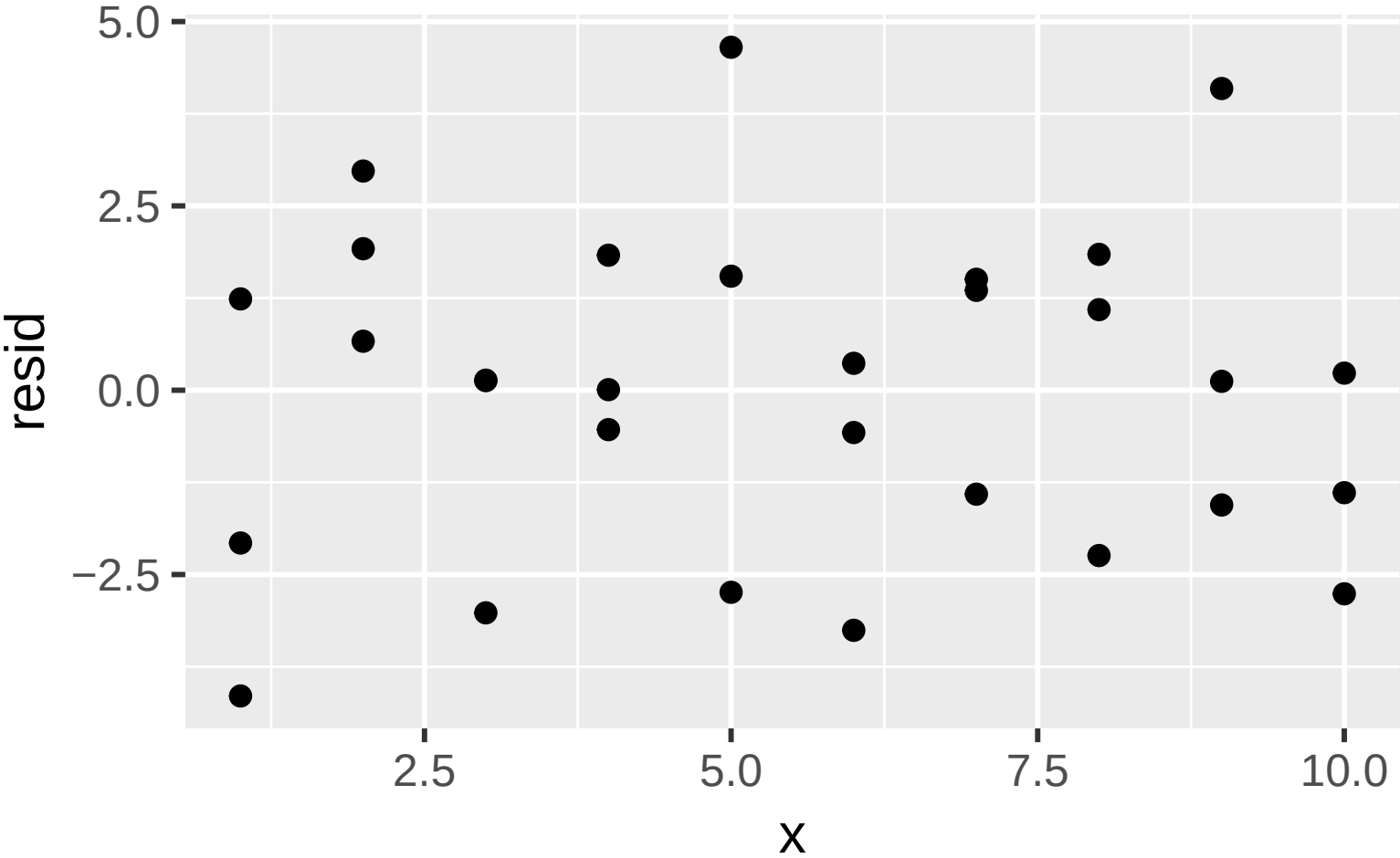
`modelr::add_residuals(fit1)`

is equivalent to:

`mutate(resid = resid(fit1))`

Plotting residuals (cont'd)

Look for "simple random scatter":



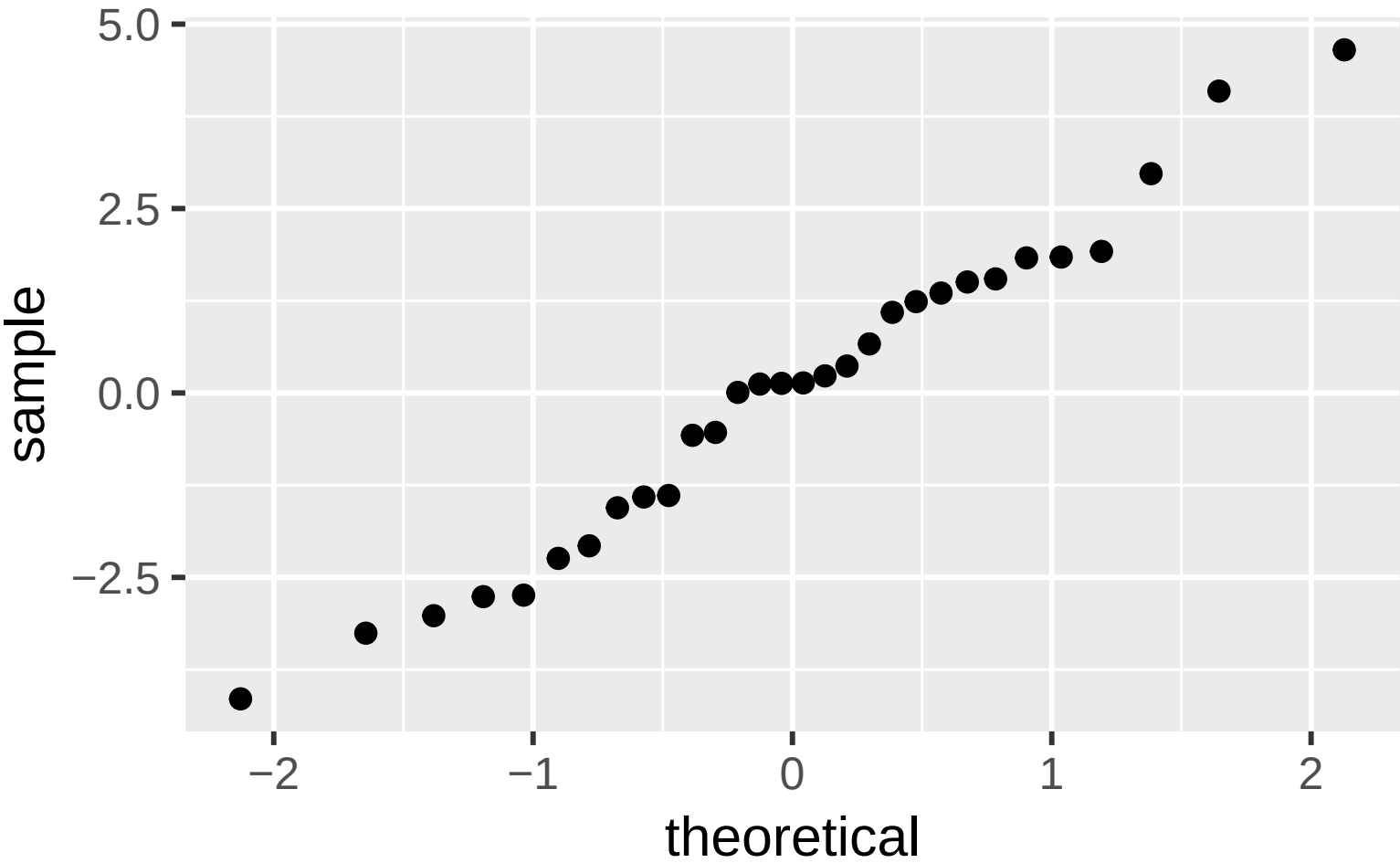
Systematic patterns indicate more relationships not captured by model.

Plotting quantiles of residuals

```
sim1 %>%
  add_residuals(fit1) %>%
  ggplot(aes(sample=resid)) +
  geom_qq()
```

Plotting quantiles of residuals (cont'd)

Look for linearity.



Deviations from linearity indicate distribution of errors is non-normal.

A note on normality

It is a common mistake when working with linear models to believe that "my data must be normally distributed".

This is **wrong**.

The assumption made by linear models is that the **errors** are normally distributed.

Let's add another predictor to our model

```
set.seed(1)
sim1b <- mutate(sim1, x2 = rnorm(n()))
fit1b <- lm(y ~ x + x2, data=sim1b)
```

```
rmse(fit1, sim1b)
```

```
## [1] 2.128181
```

```
rmse(fit1b, sim1b) # new model is more predictive!
```

```
## [1] 2.10281
```

Evaluating model quality (cont'd)

It is always possible to make a "better" model by adding more variables, **however**:

- More variables increases risk of *over-fitting*
- Does not mean the model is more "correct" or more "useful"
- Model complexity has an impact on the stability of parameter estimates
- Some measures of model quality try to account for model complexity:
  - AIC
  - BIC
  - Predictive ability with cross-validation

This is one reason why cross-validation is incredibly important when evaluating predictive models.

We will discuss cross-validation more later.

Exploratory versus confirmatory data analyis

Modeling is used differently depending on whether you are:

- Trying to *generate* a hypothesis
  - E.g., during exploratory data analysis
- Trying to *confirm* a hypothesis
  - You are doing confirmatory data analysis

Exploratory versus confirmatory data analyis (cont'd)

Statistical inference (e.g., to test a hypothesis) requires a set of assumptions about the data:

- Each observation is used *either* to generate hypotheses *or* to confirm hypotheses, but never *both*
- An observation is used only *once* to confirm hypotheses, but may be used any number of times for generating hypotheses*

In other words, data used for exploratory analysis should be independent of the data used for confirmatory analysis.

Violating either of these assumptions results in bias in your results, which will be overly optimistic.

*There are adjustments that must be made when using an observation to test multiple hypotheses at once

Partitioning data for EDA and CDA

A way to use the same dataset for both exploratory and confirmatory analysis is to partition it beforehand:

- A **training** set for visualization, exploration, and fitting as many models as you like*
- A **validation** set for comparing models and tuning data-independent model parameters
- A **testing** set for testing the final model

Alternatively, cross-validation can also be used to make use of all data for both training and testing in a safe way.

*The majority of the data should be in the training set

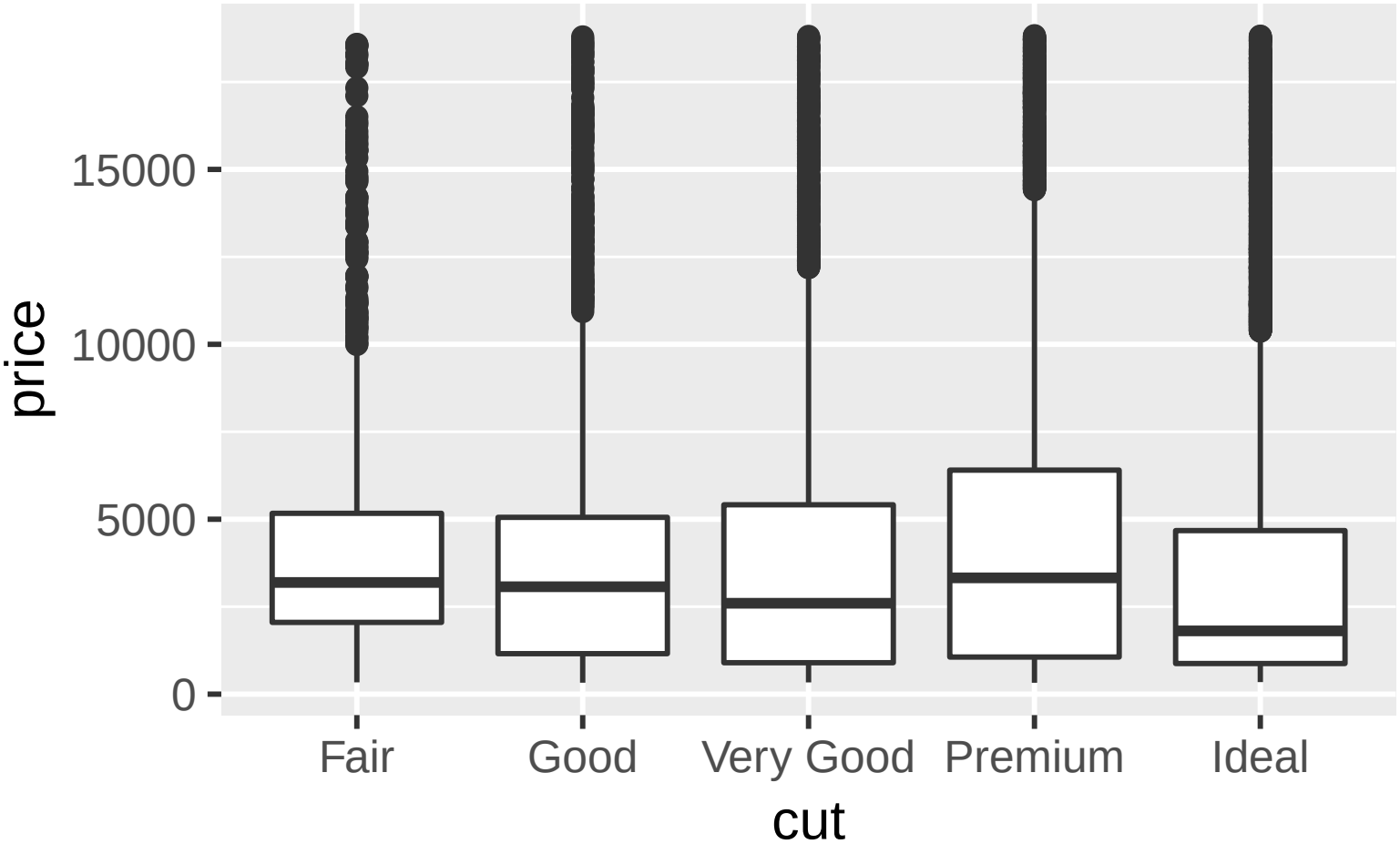
Partition the diamonds dataset

To save space, `resample_partition` returns *instructions* for partitioning the dataset (a reference to the original dataset and row indices), so we must call `as.data.frame` or `as_tibble` to get the actual subsets.

```
dmds <- resample_partition(diamonds,
                                        c(train = 0.6,
                                          valid = 0.2,
                                          test = 0.2))
dmds$train <- as_tibble(dmds$train)
```

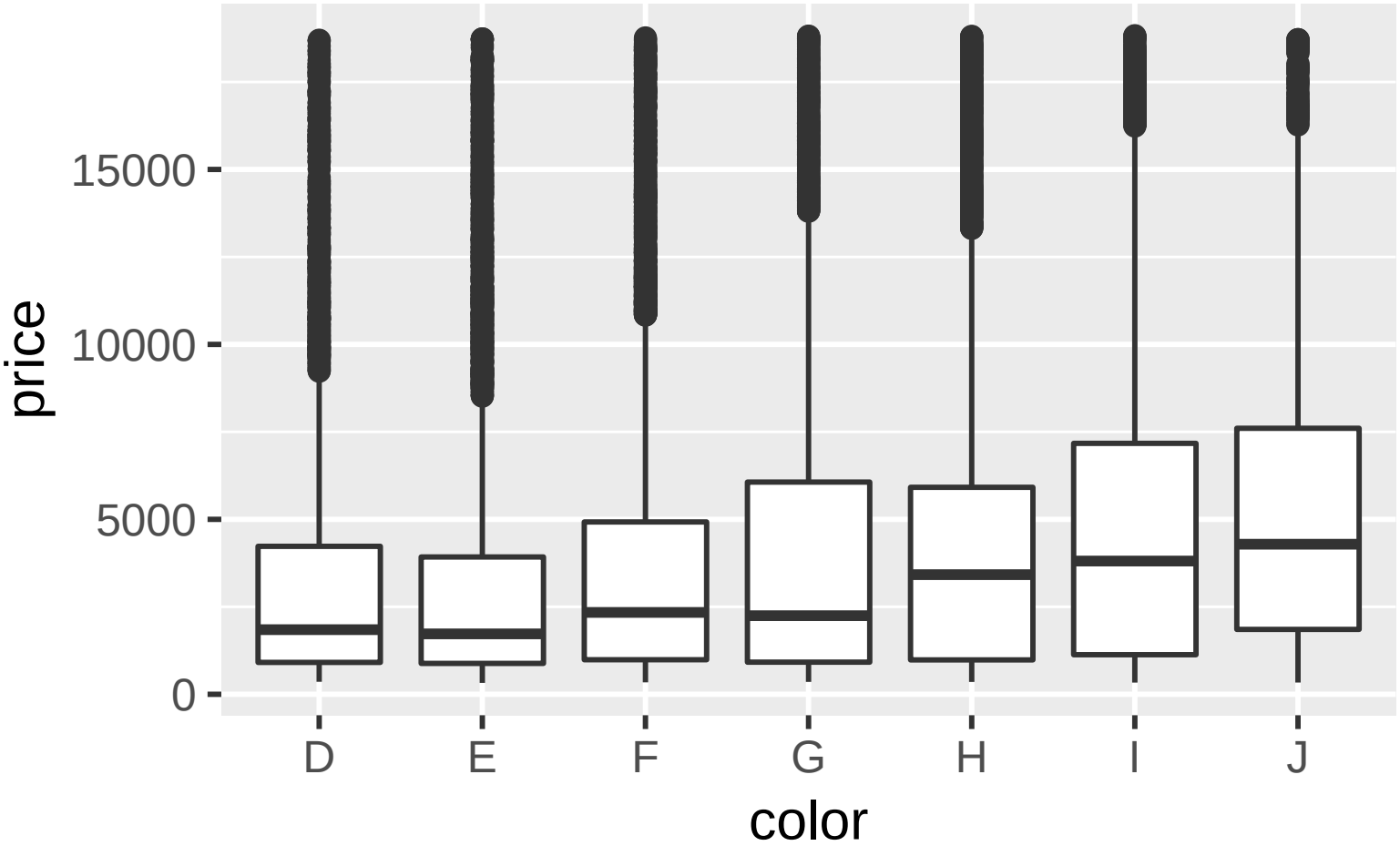
Better cut -> lower price

```
ggplot(dmds$train, aes(cut, price)) + geom_boxplot()
```



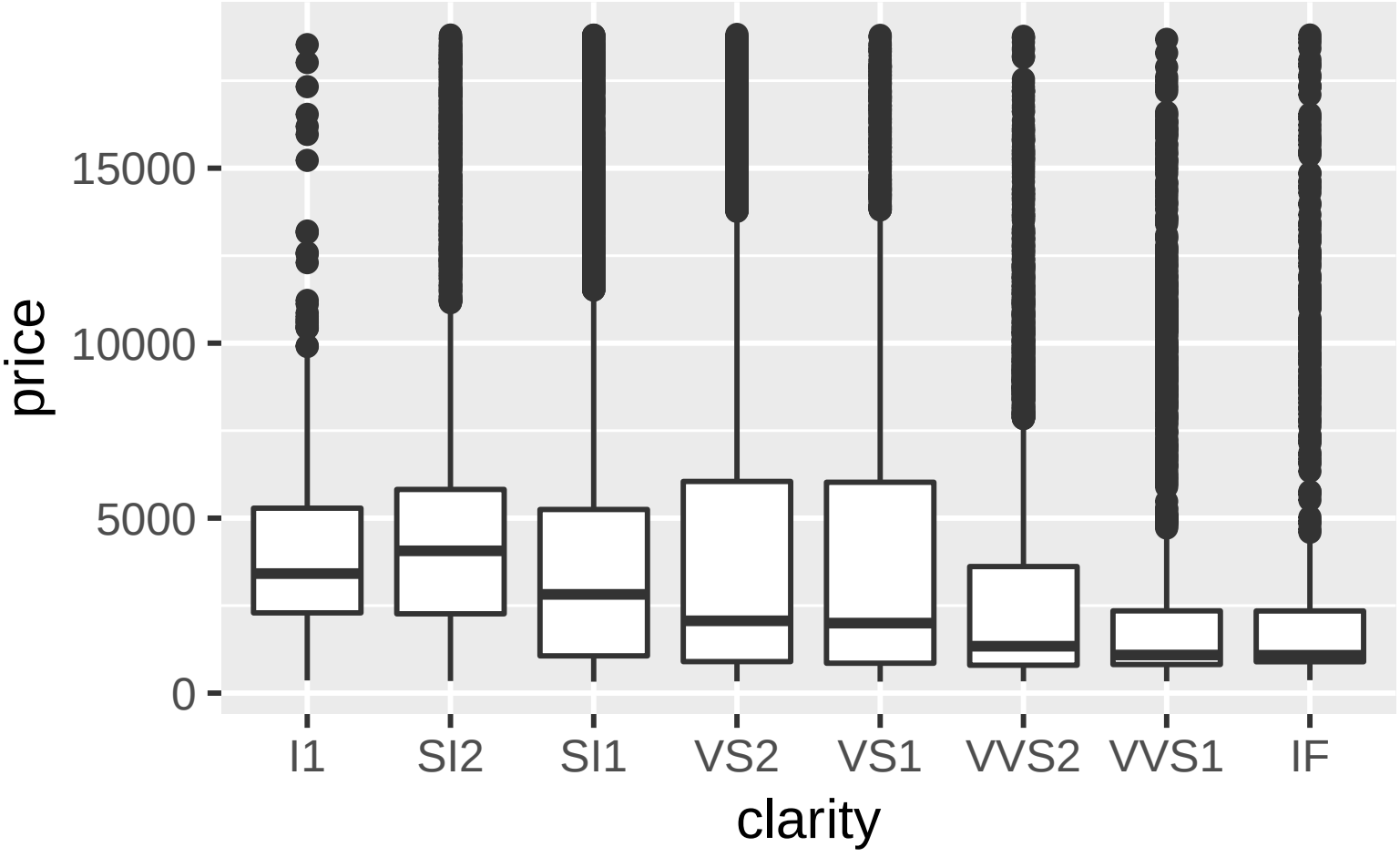
Better color -> lower price

```
ggplot(dmds$train, aes(color, price)) + geom_boxplot()
```



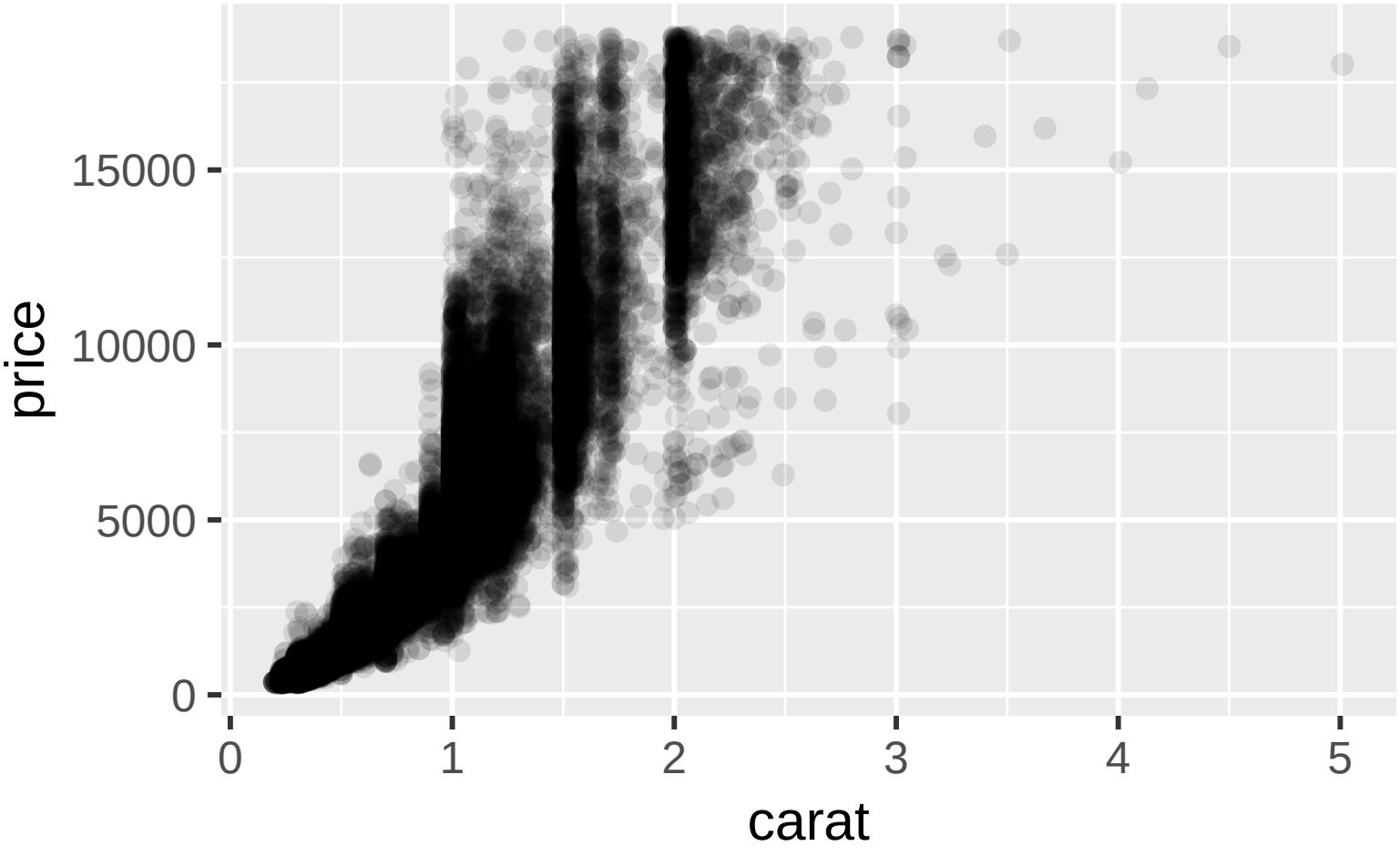
Better clarity -> lower price

```
ggplot(dmds$train, aes(clarity, price)) + geom_boxplot()
```



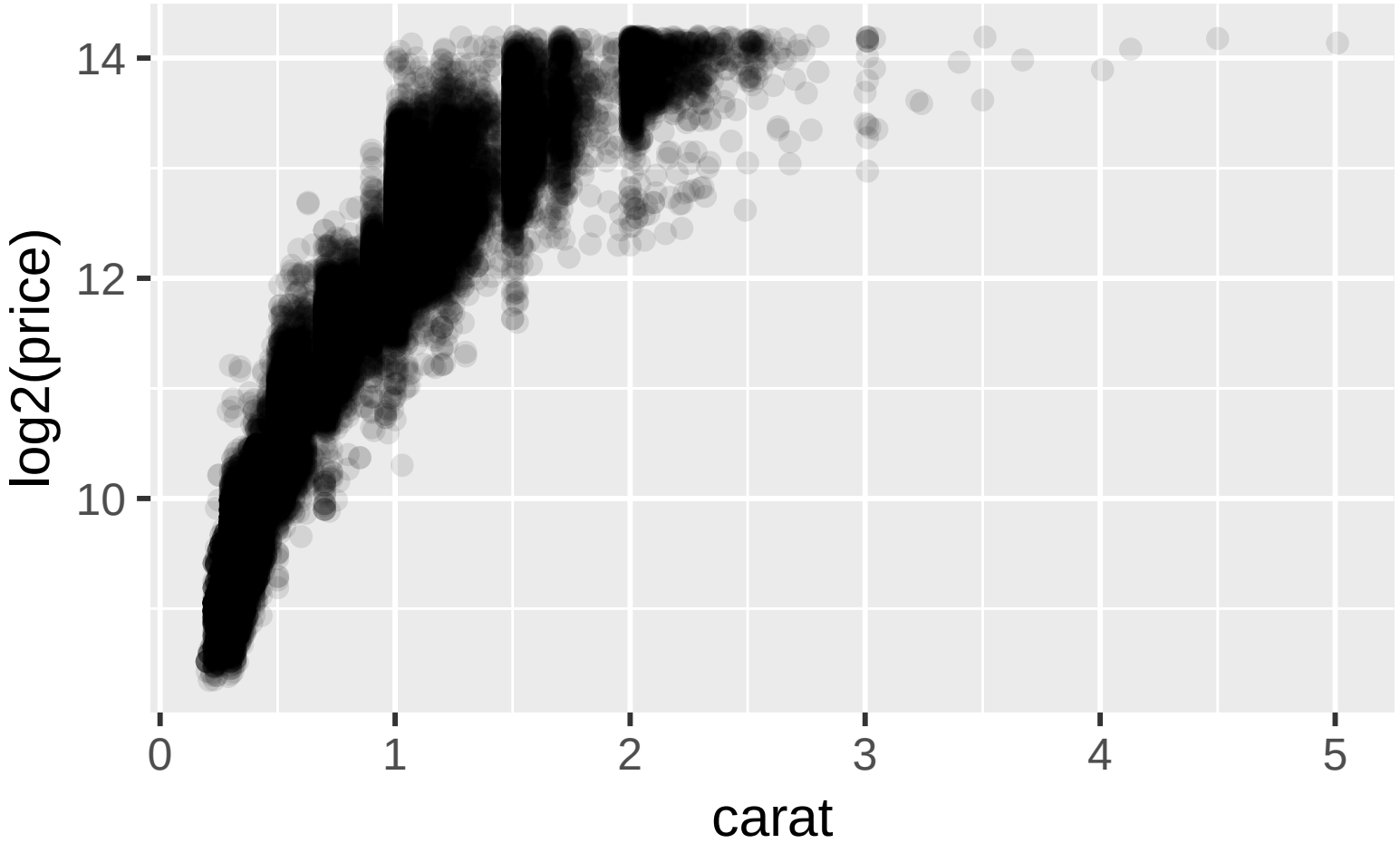
There is a relationship here but is it linear?

```
ggplot(dmds$train, aes(x=carat, y=price)) +
  geom_point(alpha=0.1)
```



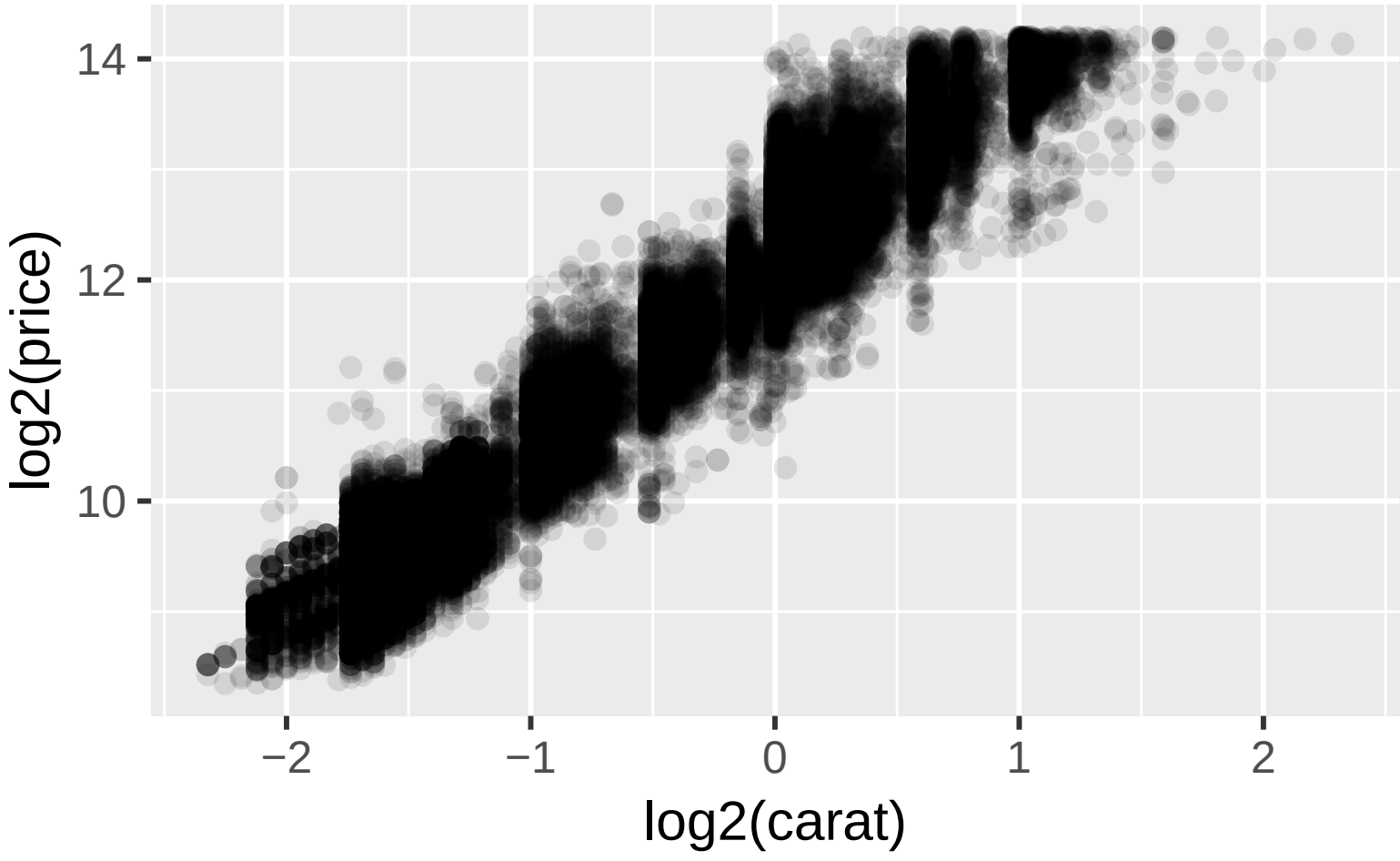
Log-transformation can help, but still nonlinear

```
ggplot(dmds$train, aes(x=carat, y=log2(price))) +
  geom_point(alpha=0.1)
```



Need to log-transform predictor too

```
ggplot(dmds$train, aes(x=log2(carat), y=log2(price))) +
  geom_point(alpha=0.1)
```



Fit a linear model

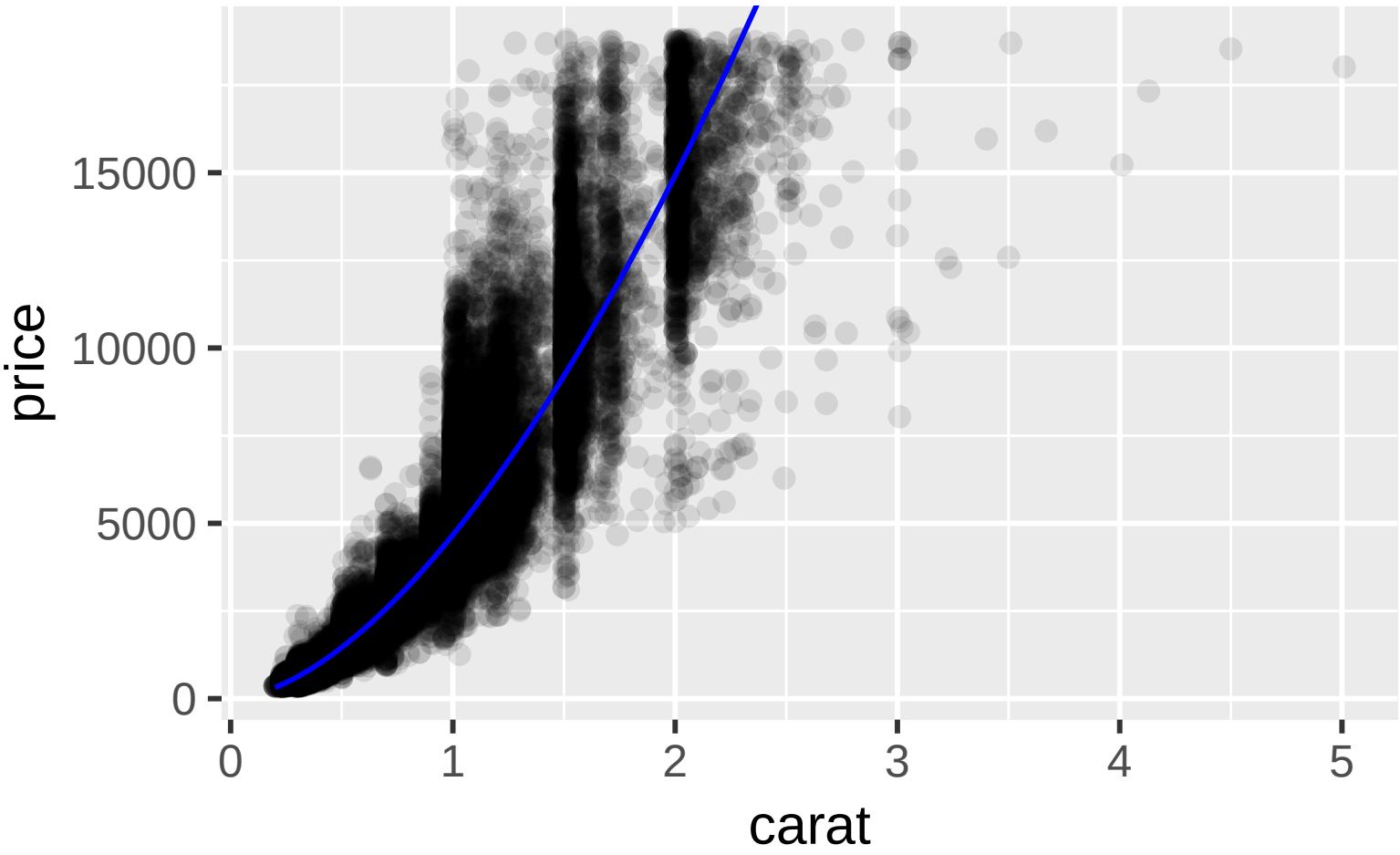
```
fit1_dmds <- lm(log2(price) ~ log2(carat), data=dmds$train)
fit1_dmds
```

```
##
## Call:
## lm(formula = log2(price) ~ log2(carat), data = dmds$train)
##
## Coefficients:
## (Intercept)  log2(carat)
##      12.189        1.676
```

- `log2(carat)` is the **predictor** or **explanatory** variable
- `log2(price)` is the **response** variable

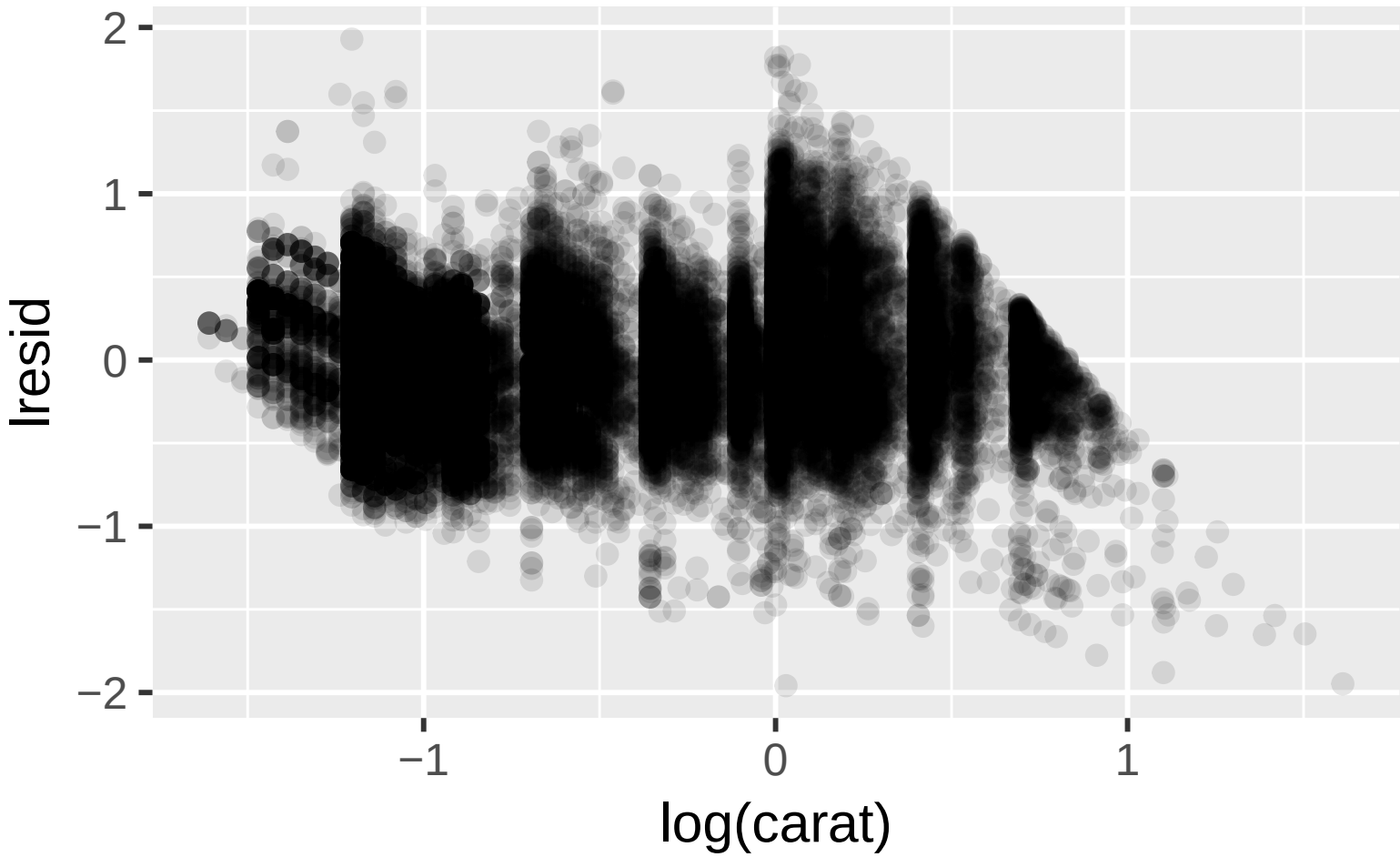
Plot the predictions

```
dmds$train %>%
  add_predictions(fit1_dmds, "lpred") %>%
  mutate(pred = 2^lpred) %>%
  ggplot(aes(x=carat)) +
  geom_point(aes(y=price), alpha=0.1) +
  geom_line(aes(y=pred), color="blue") +
  coord_cartesian(xlim=range(dmds$train$carat),
                  ylim=range(dmds$train$price))
```


price
15000
10000
5000
0
0
1
2
3
4
5
carat

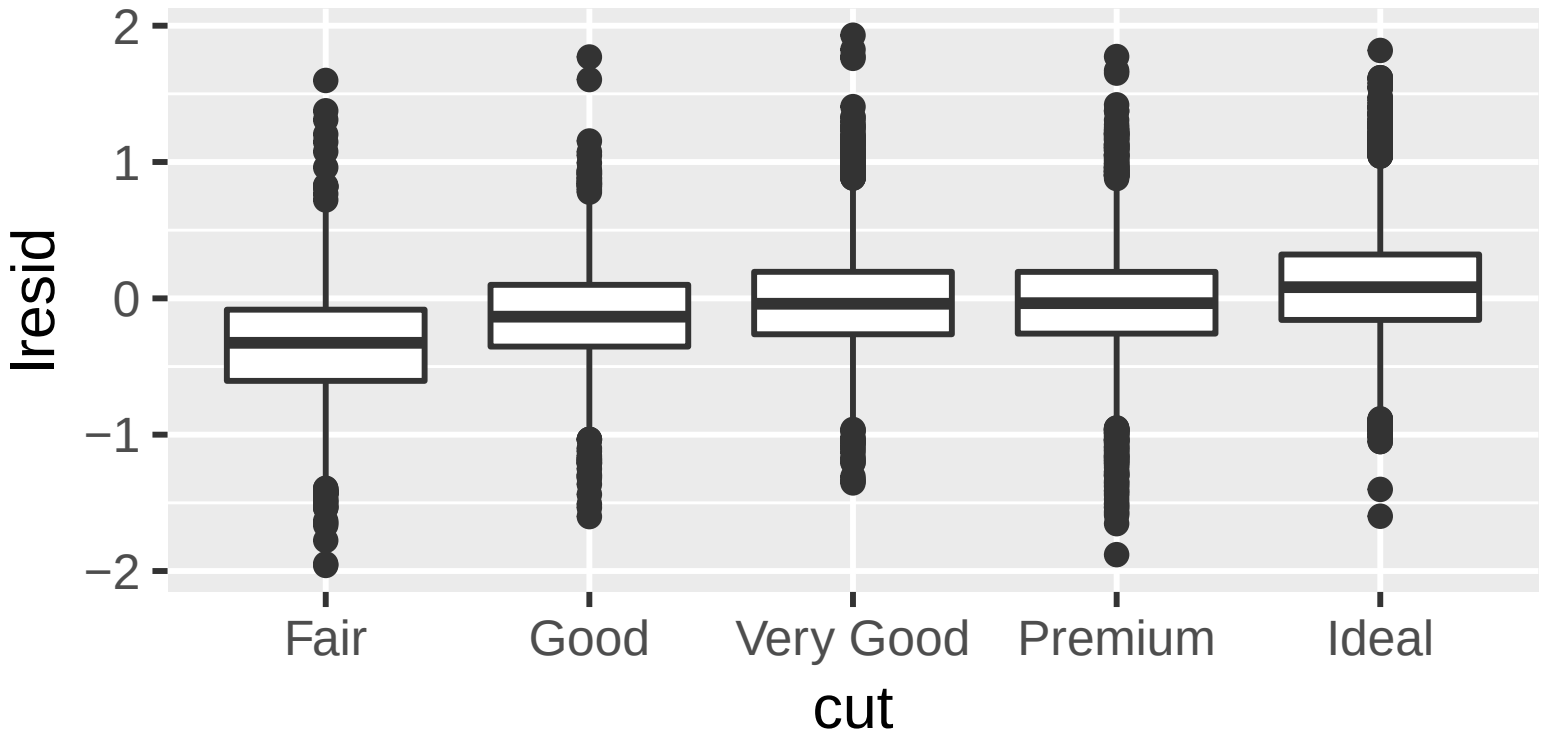
Plot the residuals

```
dmds$train %>%
  add_residuals(fit1_dmds, "lresid") %>%
  ggplot(aes(x=log(carat))) +
  geom_point(aes(y=lresid), alpha=0.1)
```


2
1
0
−1
−2
lresid
−1
0
1
log(carat)

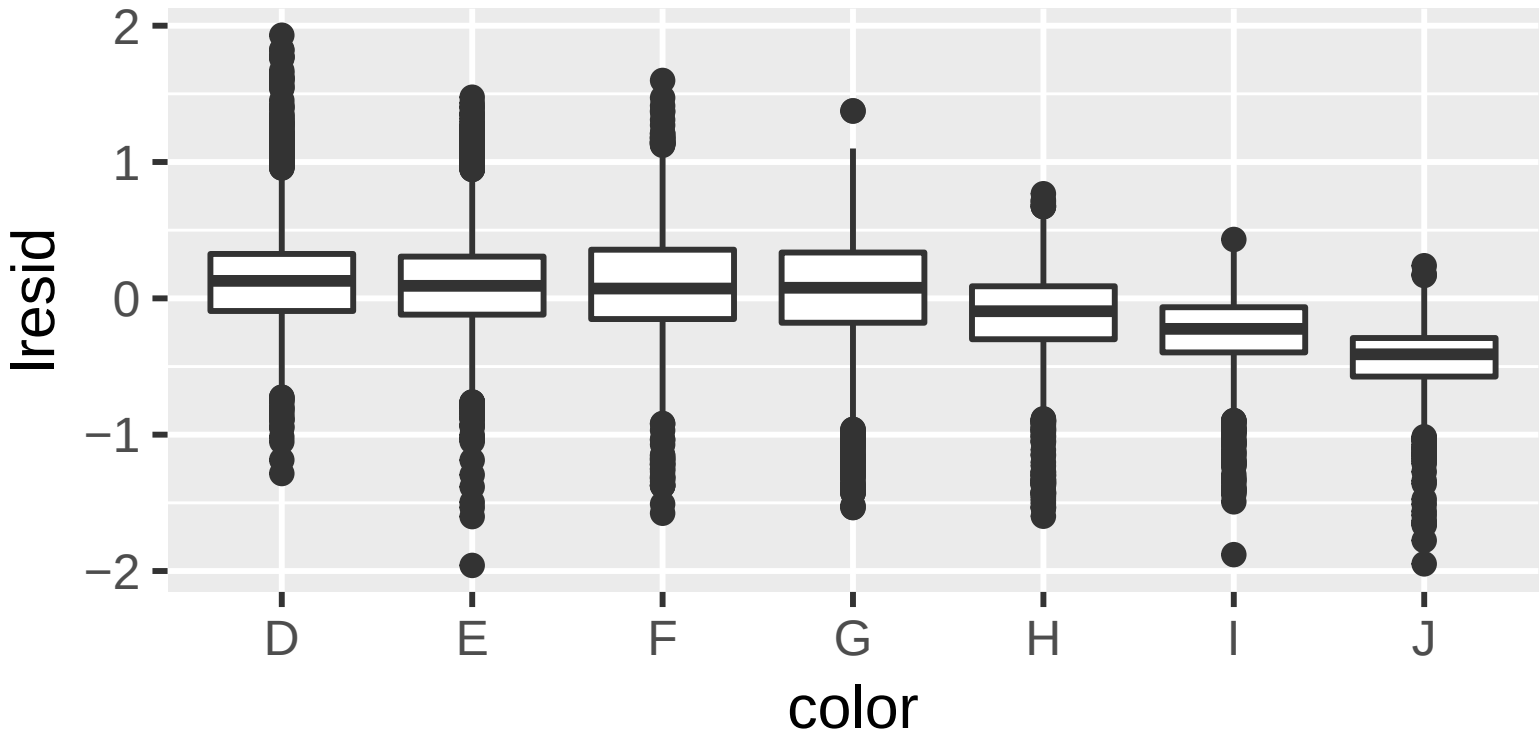
Plot the residuals vs. cut: better cut -> higher residuals

```
dmds$train %>%
  add_residuals(fit1_dmds, "lresid") %>%
  ggplot(aes(cut, lresid)) + geom_boxplot()
```



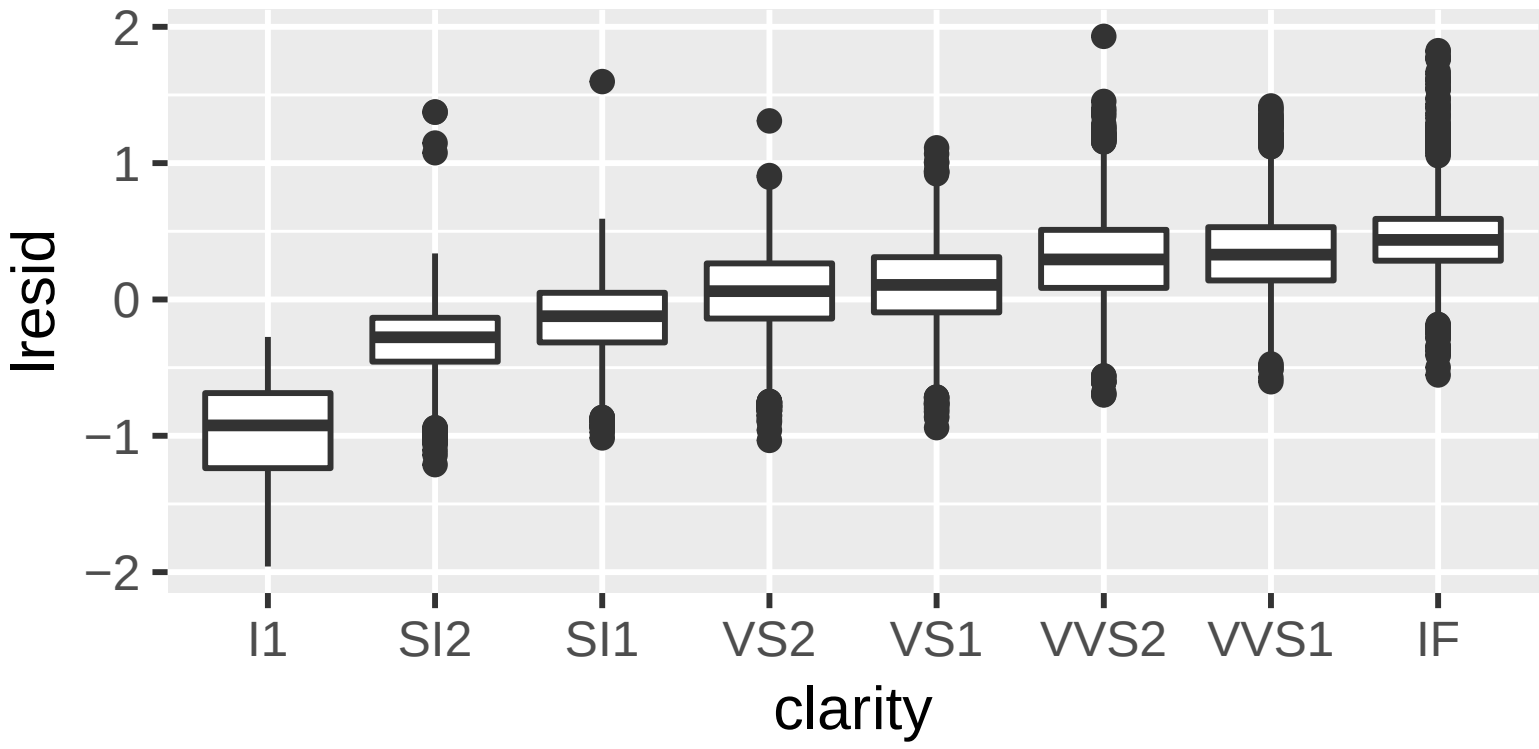
Plot the residuals vs. color: better color -> higher residuals

```
dmds$train %>%
  add_residuals(fit1_dmds, "lresid") %>%
  ggplot(aes(color, lresid)) + geom_boxplot()
```



Plot the residuals vs. clarity: better clarity -> higher residuals

```
dmds$train %>%
  add_residuals(fit1_dmds, "lresid") %>%
  ggplot(aes(clarity, lresid)) + geom_boxplot()
```



Adding predictors to the model

Although the residuals showed no systematic patterns when plotted against `carat`, we found systematic patterns when plotting the residuals against `cut`, `color`, and `clarity`.

Removing the effect of `carat` on `price` by modeling it reveals the expected relationships between price and the other variables.

We would like to add `cut`, `color`, and `clarity` to the model to improve it.

How do linear models work with categorical variables?

Simulated categorical data to model

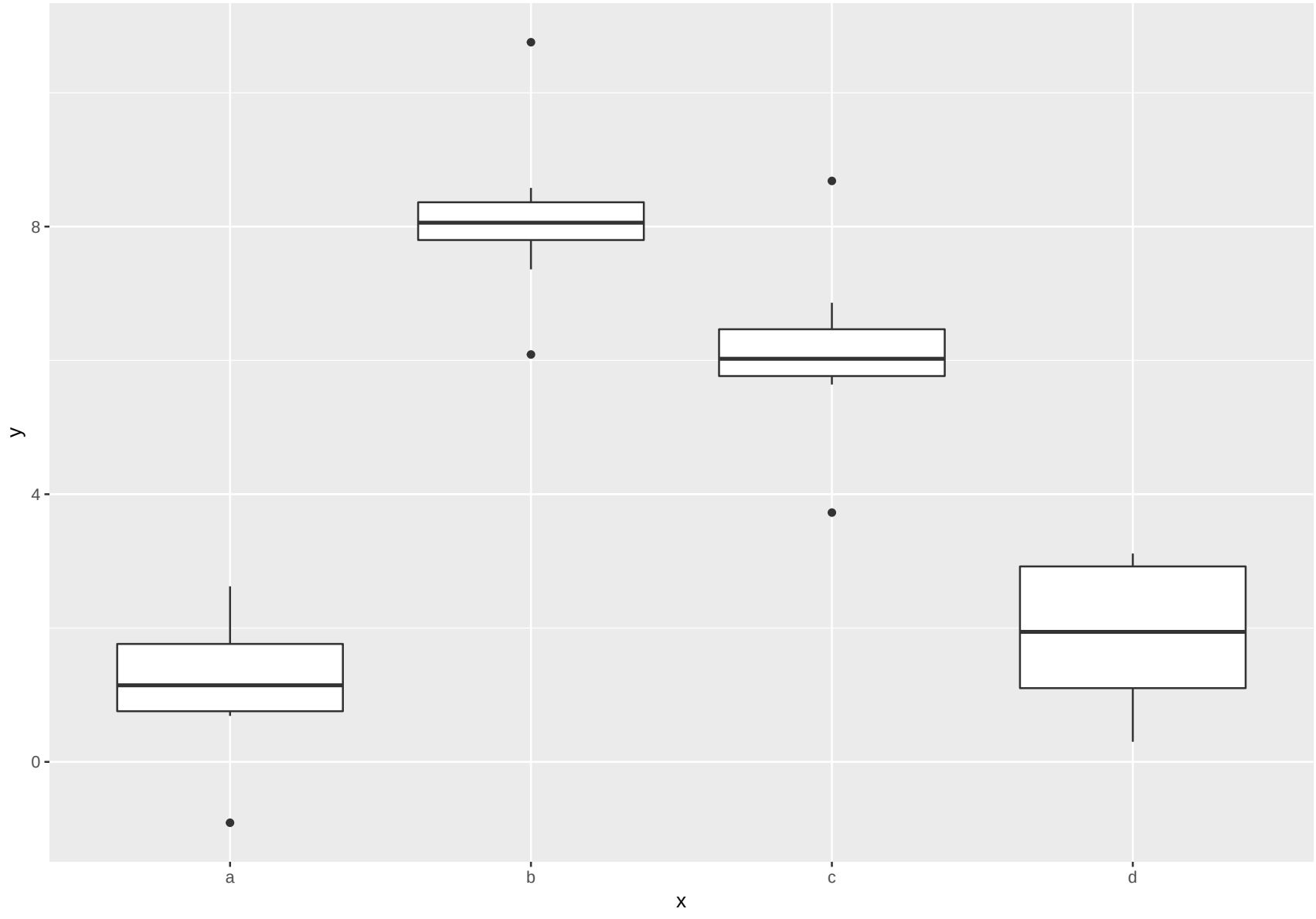
Consider the following simulated data:

```
sim2
```

```
## # A tibble: 40 x 2
##    x           y
##    <chr>   <dbl>
##  1 a       1.94
##  2 a       1.18
##  3 a       1.24
##  4 a       2.62
##  5 a       1.11
##  6 a       0.866
##  7 a      -0.910
##  8 a       0.721
##  9 a       0.687
## 10 a       2.07
## # ... with 30 more rows
```

```
ggplot(sim2, aes(x=x, y=y)) + geom_boxplot()
```

```
ggplot(sim2, aes(x=x, y=y)) + geom_boxplot()
```



How do we model `y ~ x` for categorical `x`?

Linear models use indicator variables to model categorical predictors.

Instead of

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

we get:

$$y = \beta_0 + \beta_1 x_b + \beta_2 x_c + \beta_3 x_d + \textit{error}$$

where x_b, x_c, and x_d are each 0 or 1.

One of the levels (usually the first or last, in our case "a") becomes the reference level and is not included in the indicator variables. (Why not?)

How do we model `y ~ x` for categorical `x`? (cont'd)

Mathematically, linear models are typically fit using matrices and linear algebra. The matrix form of a linear model is:

$$Y = X\beta + \epsilon$$

We can view the model matrix for `X` with `model_matrix`:

```
model_matrix(sim2, y ~ x)
```

```
## # A tibble: 40 x 4
##    `(Intercept)`    xb    xc    xd
##            <dbl> <dbl> <dbl> <dbl>
##  1             1     0     0     0
##  2             1     0     0     0
##  3             1     0     0     0
##  4             1     0     0     0
##  5             1     0     0     0
##  6             1     0     0     0
##  7             1     0     0     0
##  8             1     0     0     0
##  9             1     0     0     0
## 10             1     0     0     0
```

Why is one of the levels not included?

For our dataset `sim2`, why is the level `a` of `x` not included in the model?

It is actually included as the 'default' level, so it will correspond to the intercept. We would calculate the predicted values for each level as:

$$\hat{y}_a = \beta_0$$

$$\hat{y}_b = \beta_0 + \beta_1 x_b$$

$$\hat{y}_c = \beta_0 + \beta_1 x_c$$

$$\hat{y}_d = \beta_0 + \beta_1 x_d$$

If we included it explicitly, its column in the model matrix would be a perfect linear combination of the columns for `b`, `c`, and `d`, called **collinearity**, and the model would be **over-parameterized**.

Multicollinearity and correlation

Another key assumption of linear models is that the predicor variables are **independent** of each other.

When one predictor variable is a linear combination of other predictor variables, whether exactly or approximately (i.e., via correlation), then there is no longer a single solution for the fitted model, and there exists infinite possible combinations of model parameters that could fit the model.

This is another general danger of fitting a model with too many parameters. This is a particular problematic for datasets with many variables that are known to be correlated, such as in proteomics and genomics.

There are some methods for accounting for this problem, most of which will *not* be discussed in this class. The simplest method is to be very picky about which variables you include in your model.

Fitting a model to simulated cateorical data

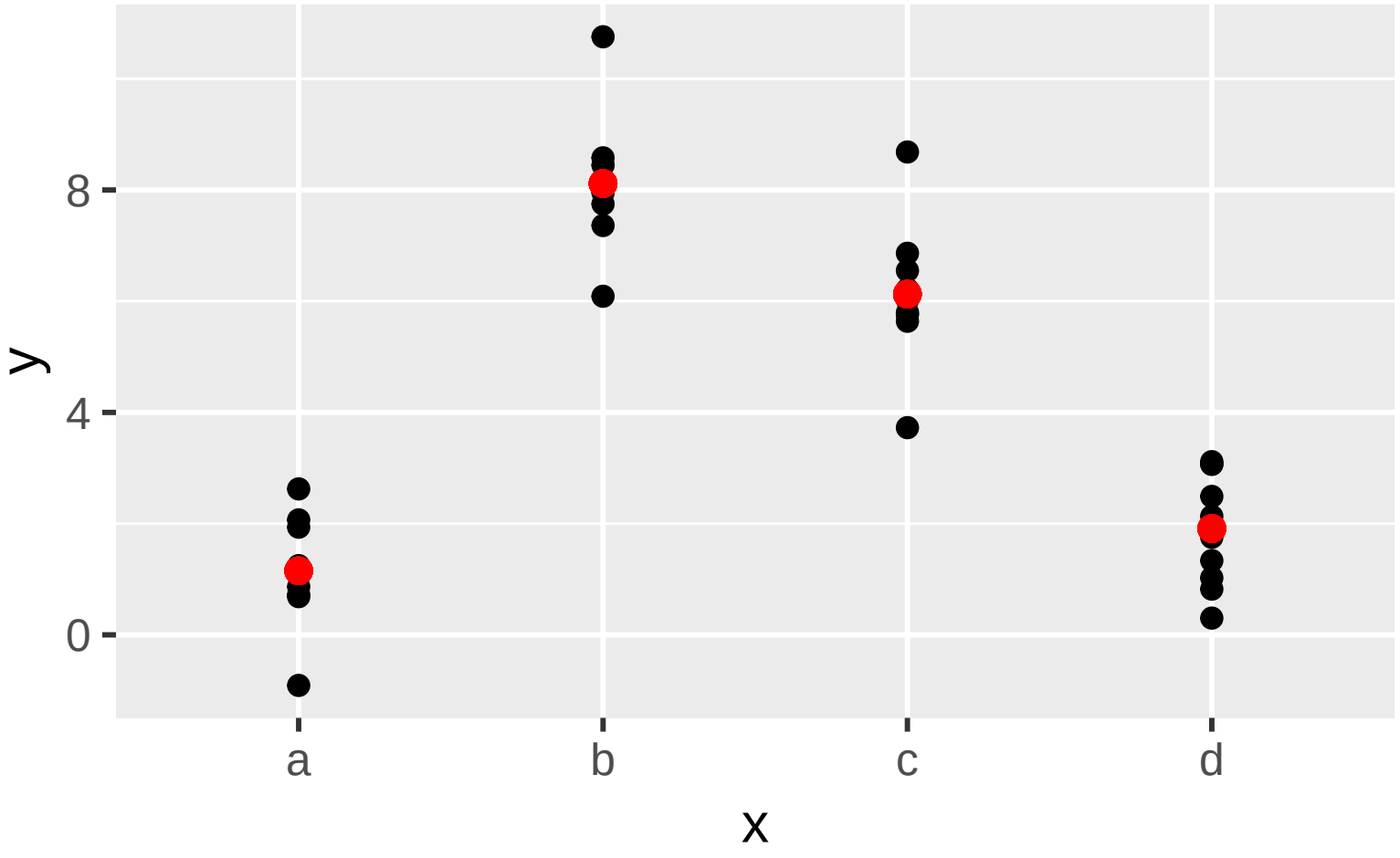
```
fit2 <- lm(y ~ x, data=sim2)
fit2
```

```
##
## Call:
## lm(formula = y ~ x, data = sim2)
##
## Coefficients:
## (Intercept)           xb           xc           xd
##      1.1522       6.9639       4.9750       0.7588
```

Plot the predictions

```
sim2 %>%
  add_predictions(fit2) %>%
  ggplot(aes(x=x)) +
  geom_point(aes(y=y)) +
  geom_point(aes(y=pred), col="red", size=2)
```

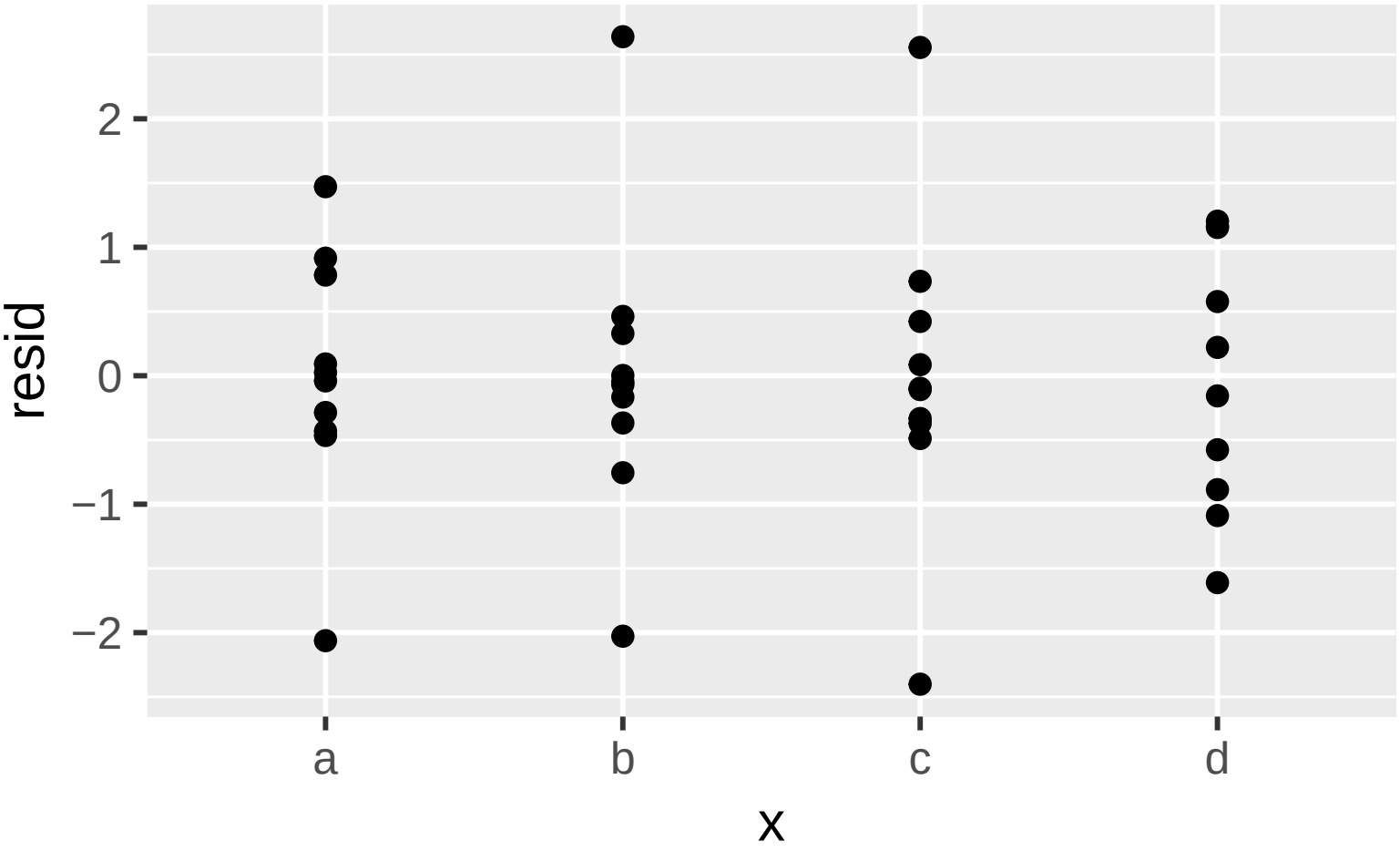
Plot the predictions (cont'd)



Plot the residuals

```
sim2 %>%
  add_residuals(fit2) %>%
  ggplot(aes(x=x)) +
  geom_point(aes(y=resid))
```

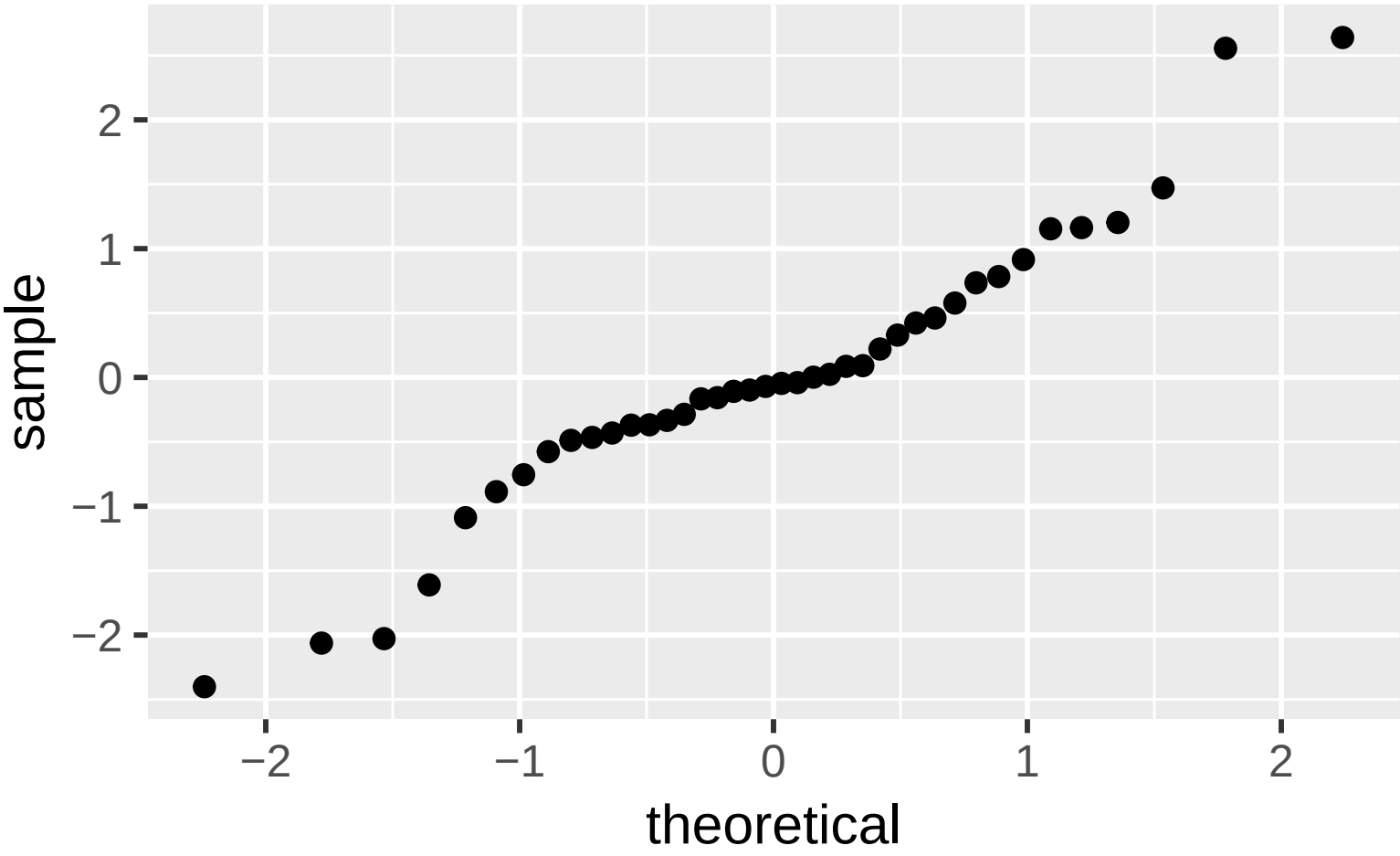
Plot the residuals (cont'd)



Plot the quantiles of residuals

```
sim2 %>%
  add_residuals(fit2) %>%
  ggplot(aes(sample=resid)) +
  geom_qq()
```

Plot the quantiles of residuals (cont'd)



Adding varibles to our model for `diamonds`

Now we can add `cut`, `color`, and `clarity` to the model:

```
fit2_dmds <- lm(log2(price) ~
                  log2(carat) + cut + color + clarity,
                data=dmds$train)
```

- `log2(carat)`, `cut`, `color`, and `clarity` are the **predictor** variables
- `log2(price)` is the **response** variable

A more complex model for diamonds

```
fit2_dmds
```

```
##
## Call:
## lm(formula = log2(price) ~ log2(carat) + cut + color + clarit
##     data = dmds$train)
##
## Coefficients:
## (Intercept)  log2(carat)         cut.L         cut.Q         cut
##  12.1987395    1.8842495    0.1705685   -0.0472801    0.01680
##       cut^4      color.L      color.Q      color.C      color
##   0.0003414   -0.6334402   -0.1411417   -0.0257943    0.01562
##     color^5      color^6    clarity.L    clarity.Q    clarity
##  -0.0045970    0.0008477    1.3300492   -0.3686817    0.19586
##   clarity^4    clarity^5    clarity^6    clarity^7
##  -0.1028432    0.0430714   -0.0005360    0.0500150
```

Predictive accuracy vs simple model

```
rmse(fit1_dmds, dmds$valid)
```

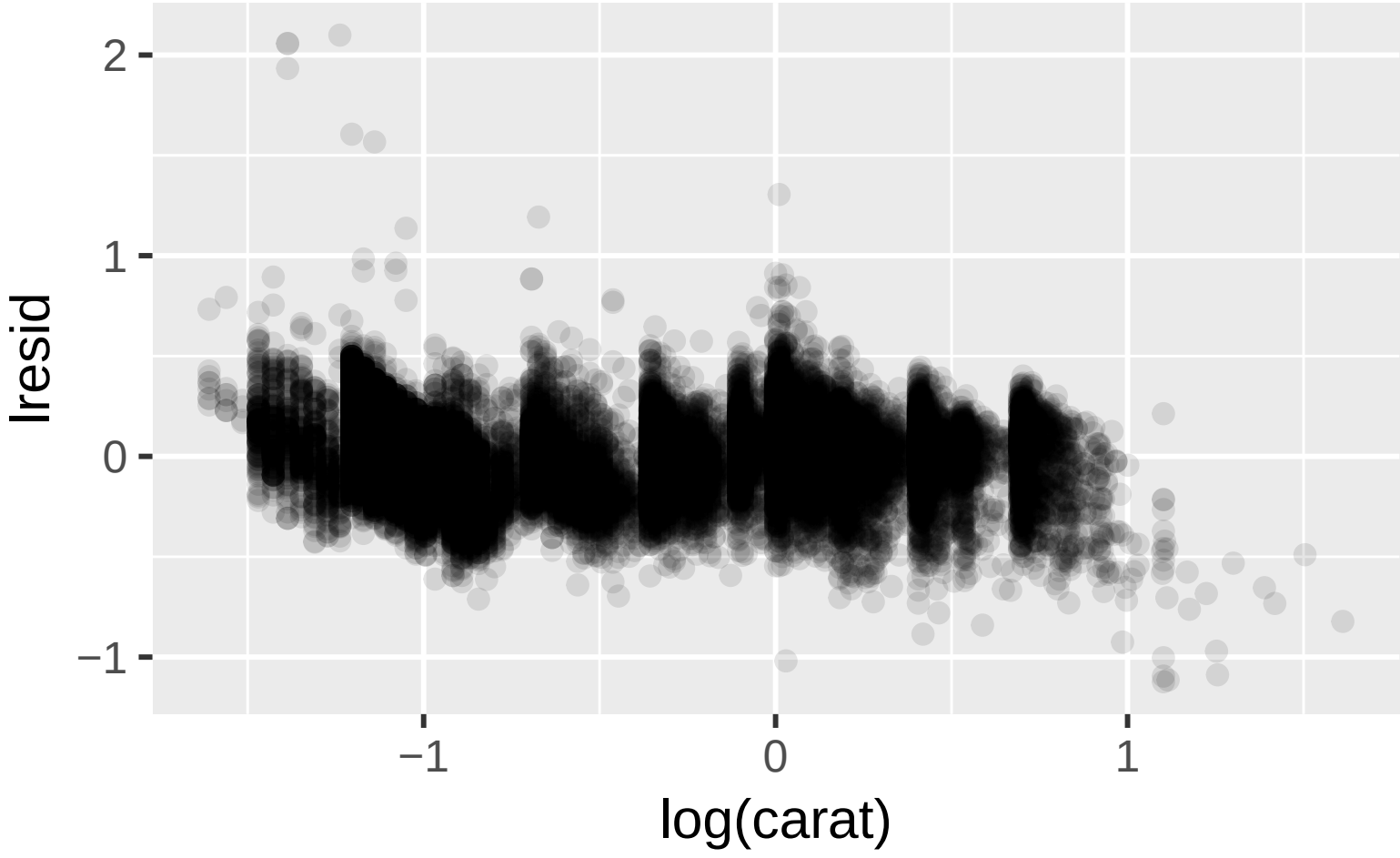
```
## [1] 0.3779044
```

```
rmse(fit2_dmds, dmds$valid)
```

```
## [1] 0.1943445
```

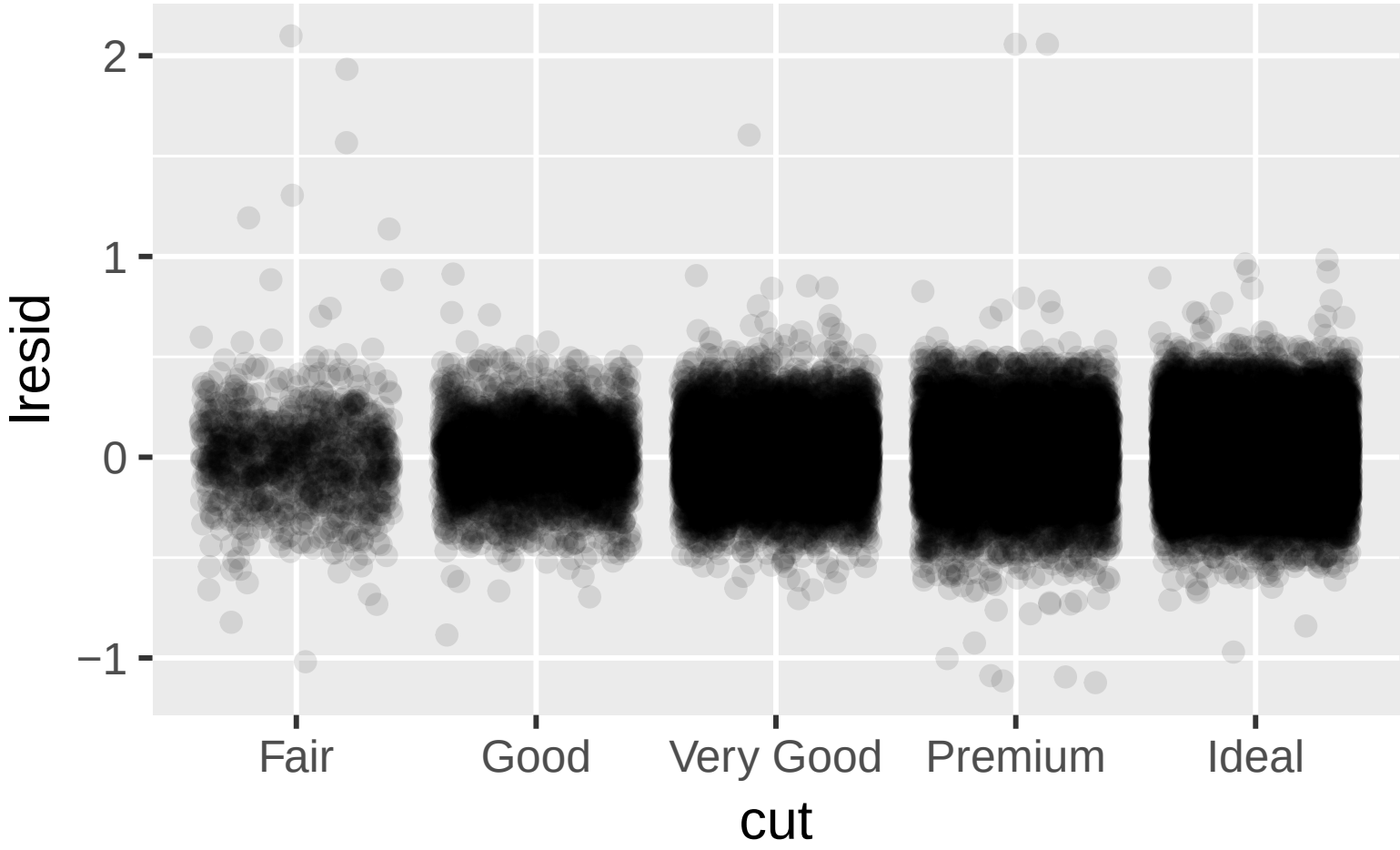
Plot residuals vs. carat

```
dmds$train %>%
  add_residuals(fit2_dmds, "lresid") %>%
  ggplot(aes(x=log(carat), y=lresid)) + geom_point(alpha=0.1)
```



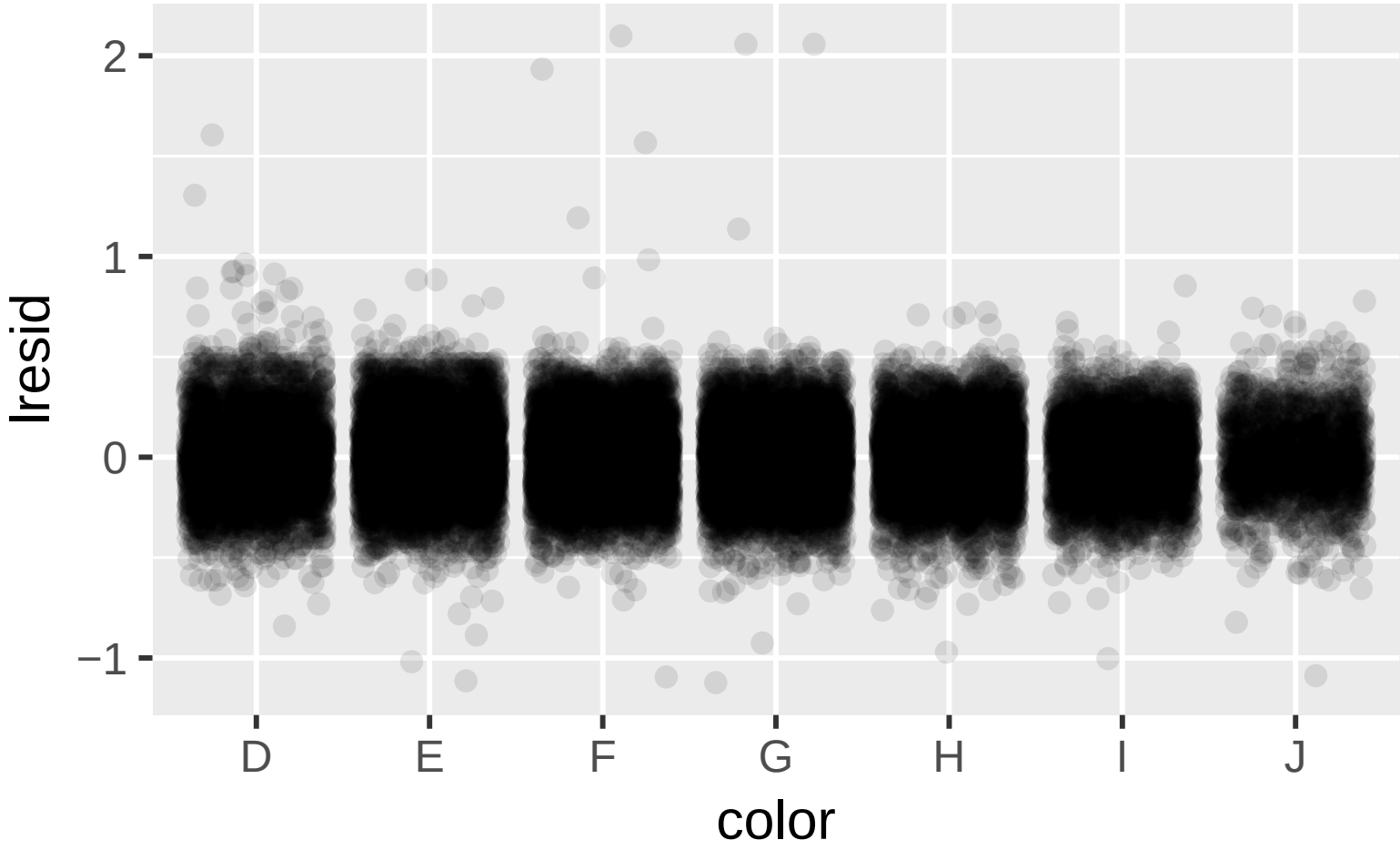
Plot residuals vs. cut

```
dmds$train %>%
  add_residuals(fit2_dmds, "lresid") %>%
  ggplot(aes(x=cut, y=lresid)) +
  geom_point(position="jitter", alpha=0.1)
```


2
1
0
−1
lresid
Fair
Good
Very Good
Premium
Ideal
cut

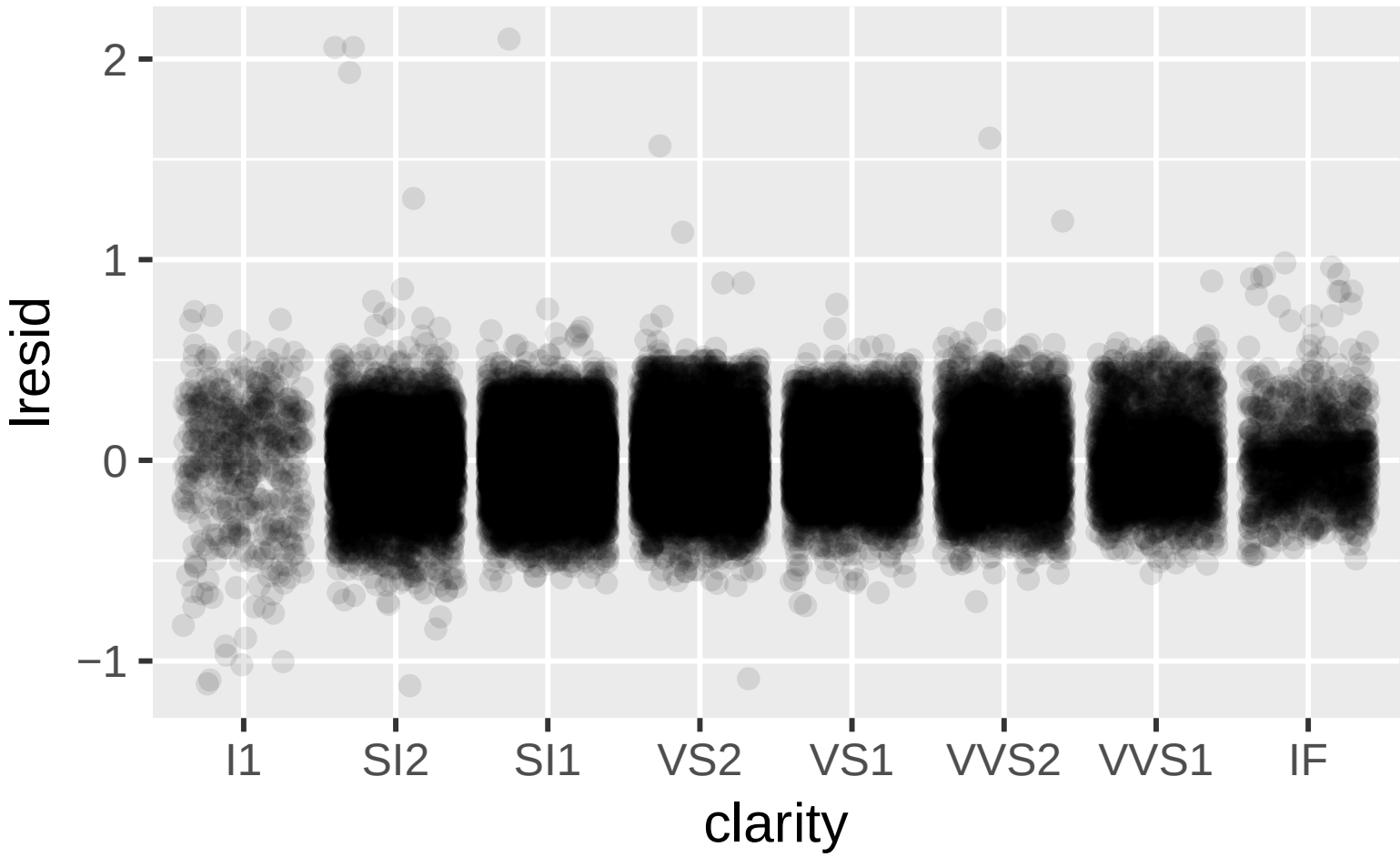
Plot residuals vs. color

```
dmds$train %>%
  add_residuals(fit2_dmds, "lresid") %>%
  ggplot(aes(x=color, y=lresid)) +
  geom_point(position="jitter", alpha=0.1)
```


2
1
0
−1
lresid
D
E
F
G
H
I
J
color

Plot residuals vs. clarity

```
dmds$train %>%
  add_residuals(fit2_dmds, "lresid") %>%
  ggplot(aes(x=clarity, y=lresid)) +
  geom_point(position="jitter", alpha=0.1)
```


lresid
2
1
0
−1
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF
clarity

Summary of model

```
summary(fit2_dmds)
```

```
##
## Call:
## lm(formula = log2(price) ~ log2(carat) + cut + color + clarity,
##     data = dmds$train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.12295 -0.12436  0.00051  0.11980  2.09916
##
## Coefficients:
##                Estimate Std. Error  t value Pr(>|t|)
## (Intercept) 12.1987395  0.0021721 5616.162  < 2e-16 ***
## log2(carat)  1.8842495  0.0014533 1296.539  < 2e-16 ***
## cut.L        0.1705685  0.0043912   38.843  < 2e-16 ***
## cut.Q       -0.0472801  0.0038657  -12.231  < 2e-16 ***
## cut.C        0.0168021  0.0033506    5.015 5.34e-07 ***
## cut^4        0.0003414  0.0026774    0.128    0.899
## color.L     -0.6334402  0.0037606 -168.441  < 2e-16 ***
## color.Q     -0.1411417  0.0034502  -40.908  < 2e-16 ***
## color.C     -0.0257943  0.0032360   -7.971 1.62e-15 ***
```