THE UNIVERSITY OF TEXAS AT AUSTIN
McCOMBS SCHOOL OF BUSINESS

Training and test sets

Lecture 22

STA 371G

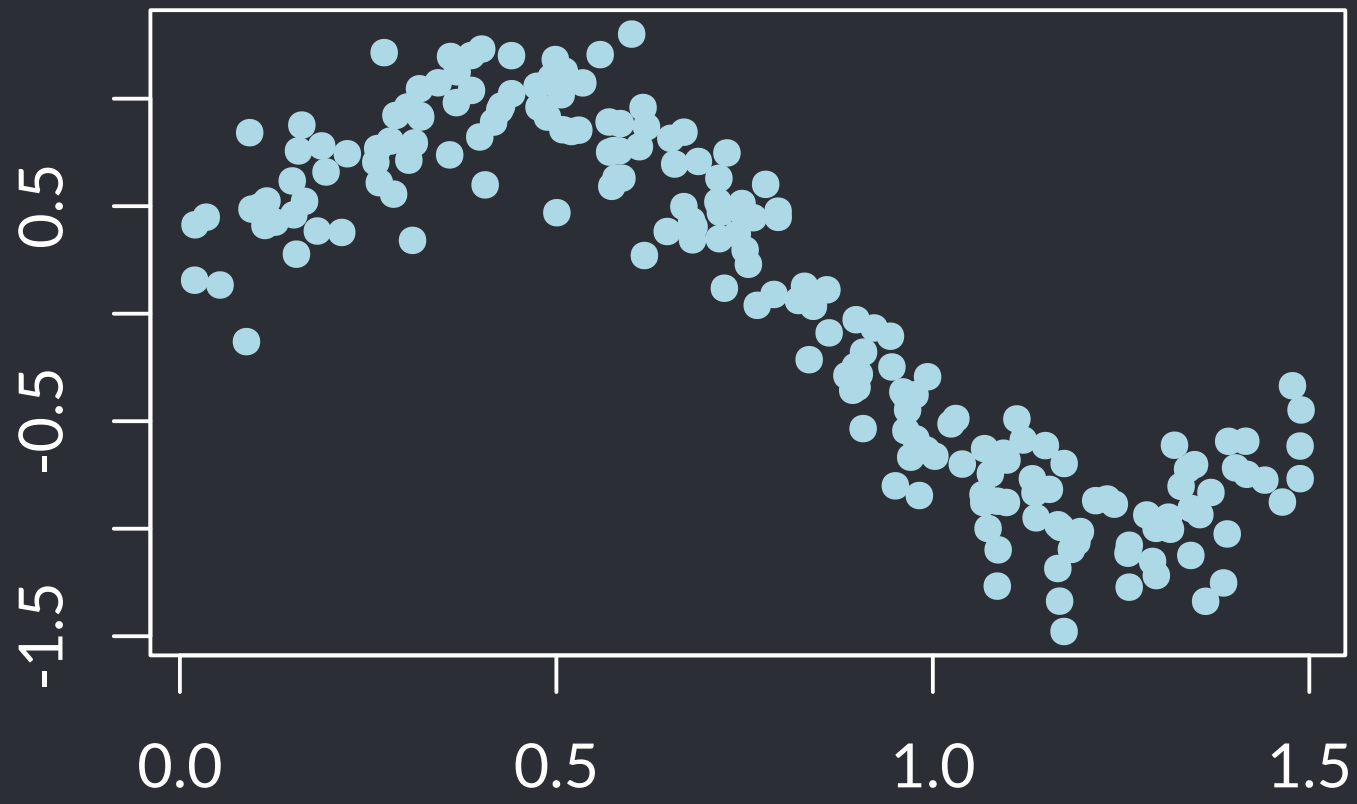
1. Overfitting

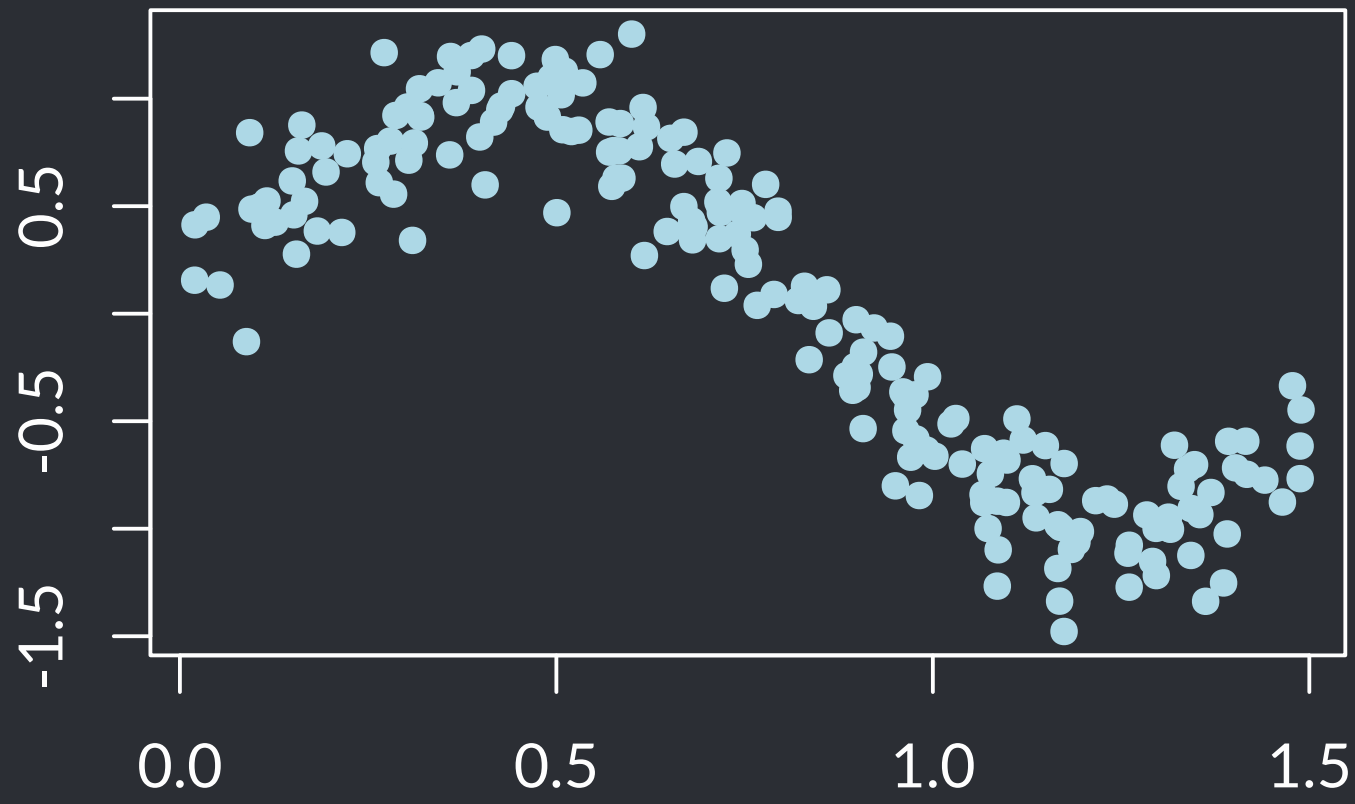
2. Using *p*-values responsibly to avoid overfitting and *p*-hacking

3. Training and test sets

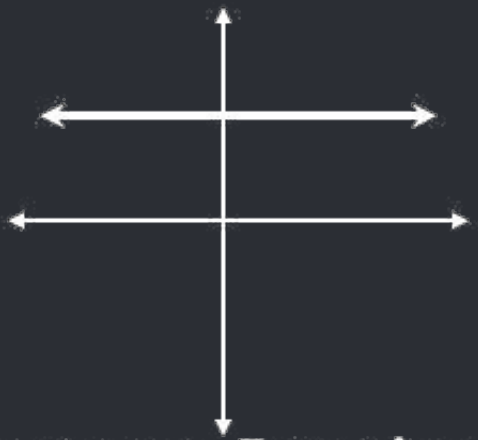
Overfitting

- A related problem to *p*-hacking is the issue of overfitting: creating a model that fits your *sample* very well but does not generalize well to the larger *population*.
- In other words, an overfit model is one where the R^2 (for linear regression) or prediction accuracy (for logistic regression) is high, but the model will not work as well as expected when given new data.
- Let's consider a simple problem of predicting Y from one X, and fitting a polynomial curve to the data.


0.5
-0.5
-1.5
0.0
0.5
1.0
1.5

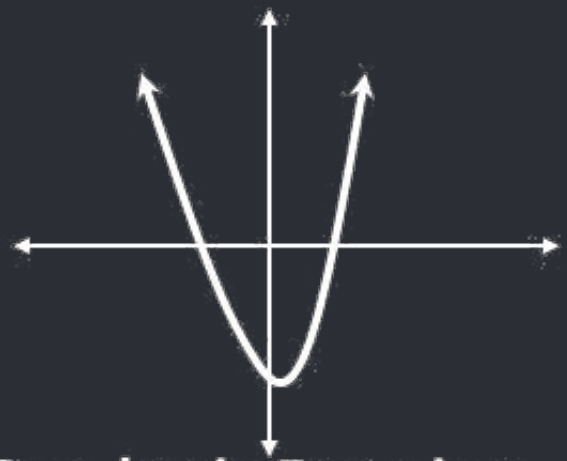


As we increase the degree (the highest power of x) of the polynomial, the fit improves, since the curve can zig and zag to capture more of the idiosyncrasies of the sample.

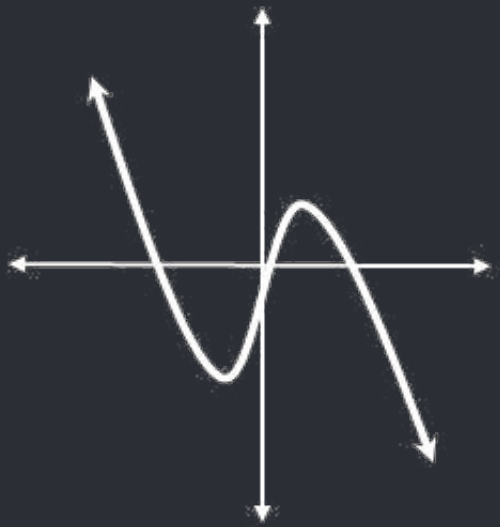


Constant Function
(degree = 0)

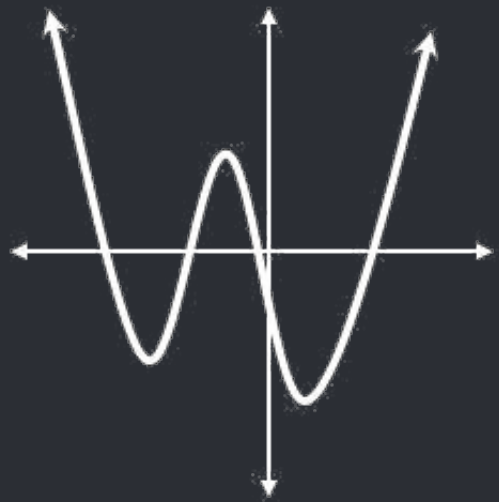
Linear Function
(degree = 1)



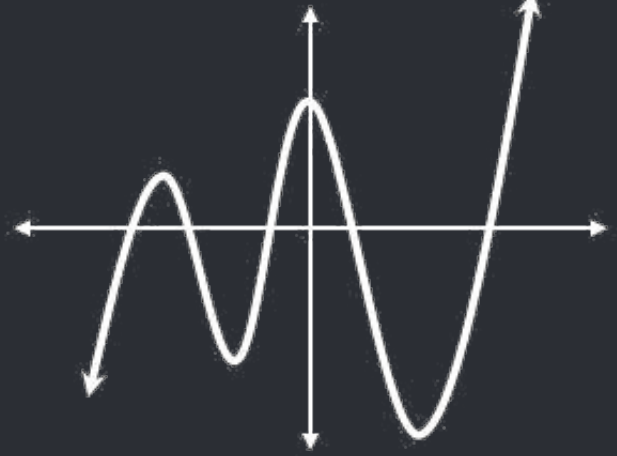
Quadratic Function
(degree = 2)



Cubic Function
(deg. = 3)

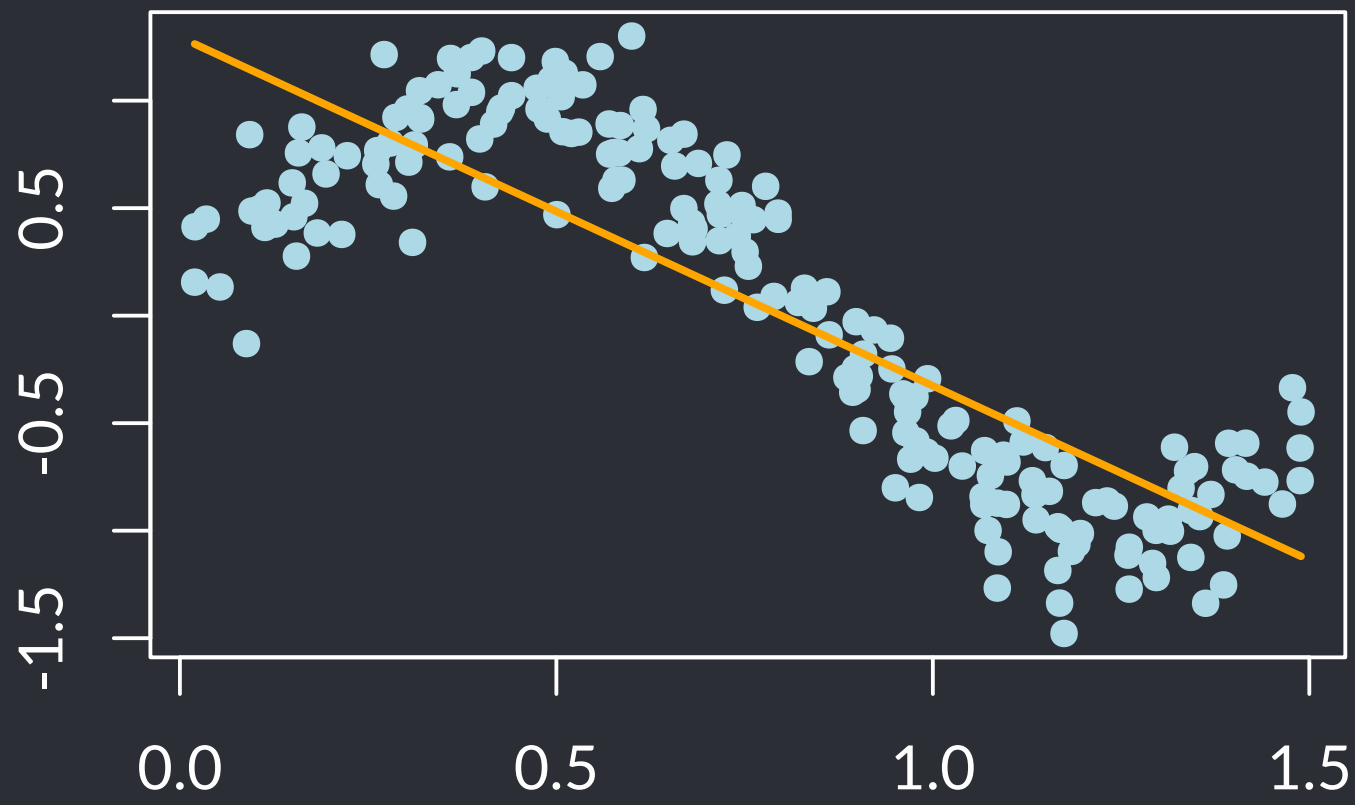


Quartic Function
(deg. = 4)



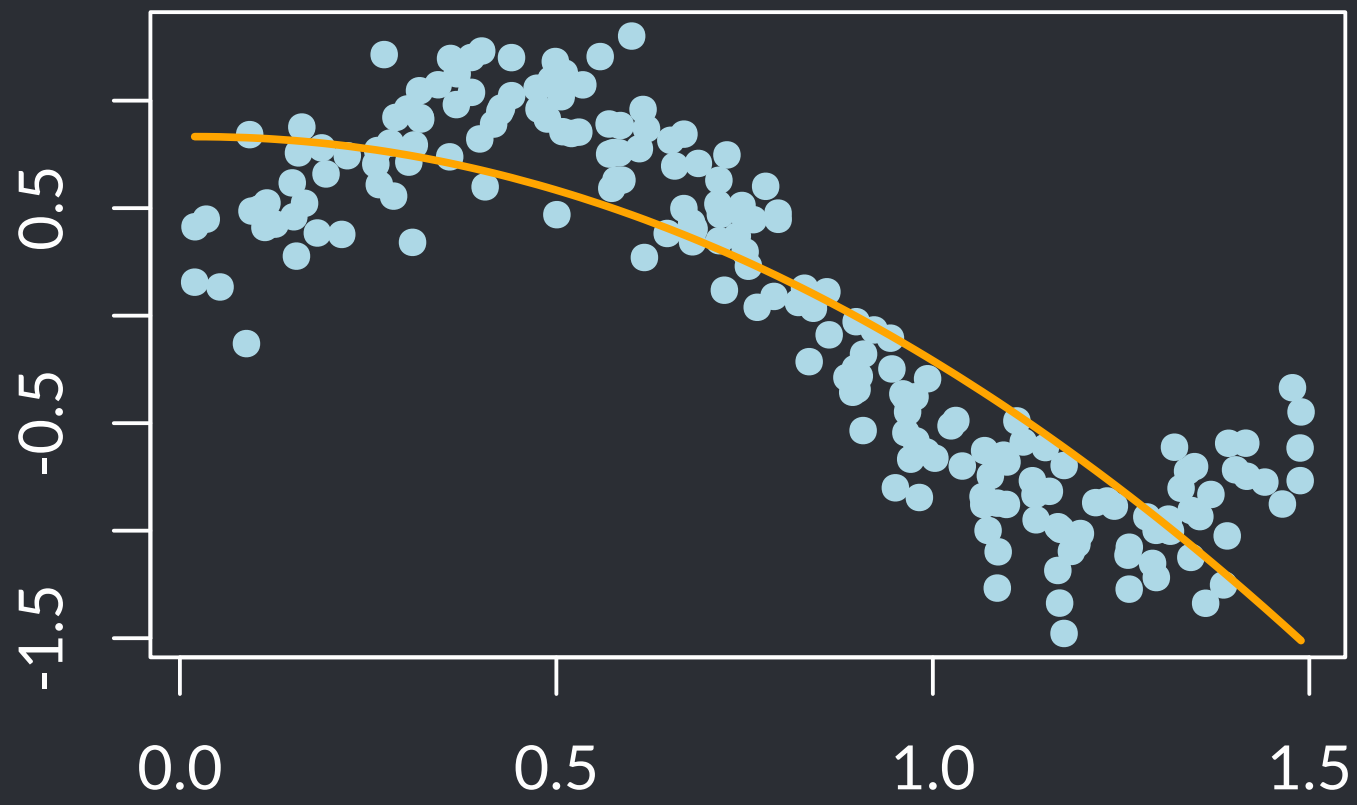
Quintic Function
(deg. = 5)

Degree 1 polynomial



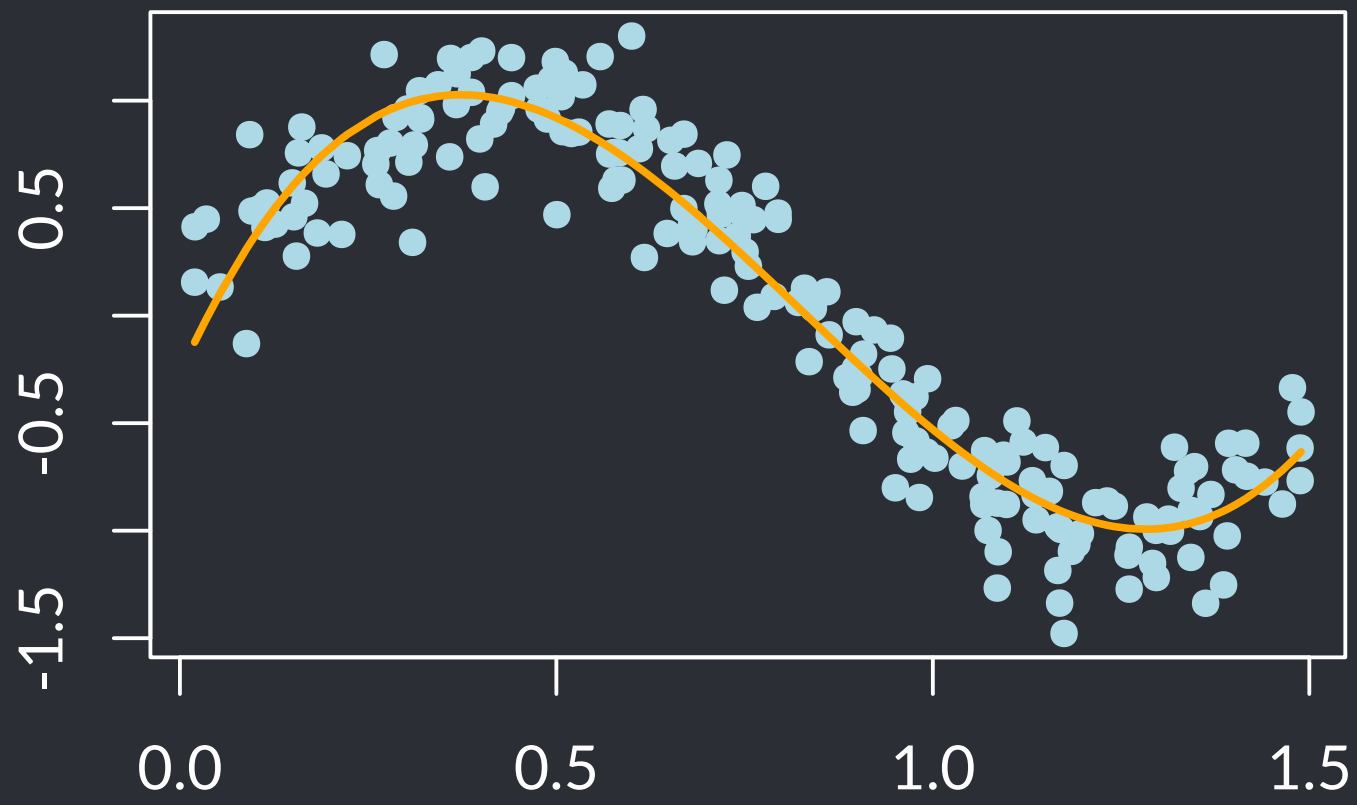
average absolute prediction error = 0.3438

Degree 2 polynomial



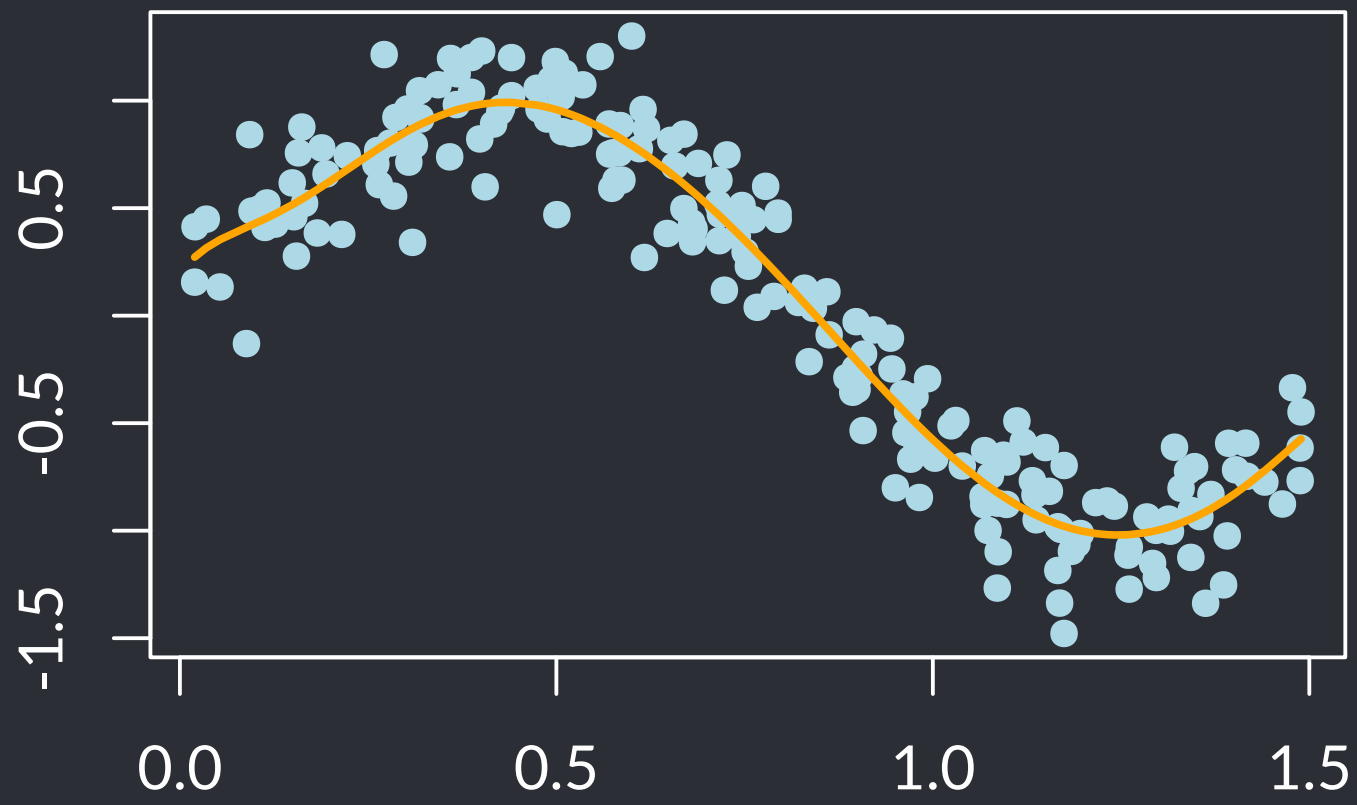
average absolute prediction error = 0.3097

Degree 3 polynomial



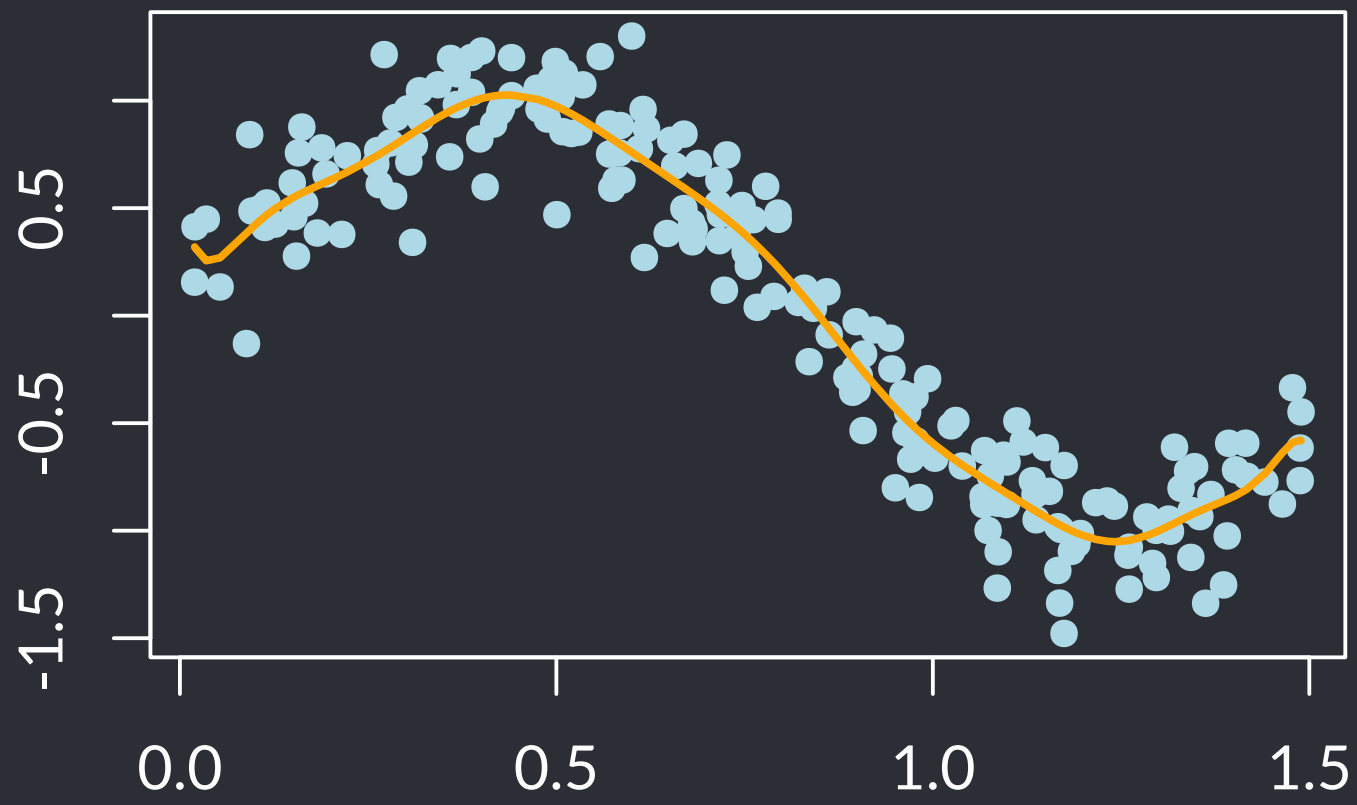
average absolute prediction error = 0.1696

Degree 10 polynomial



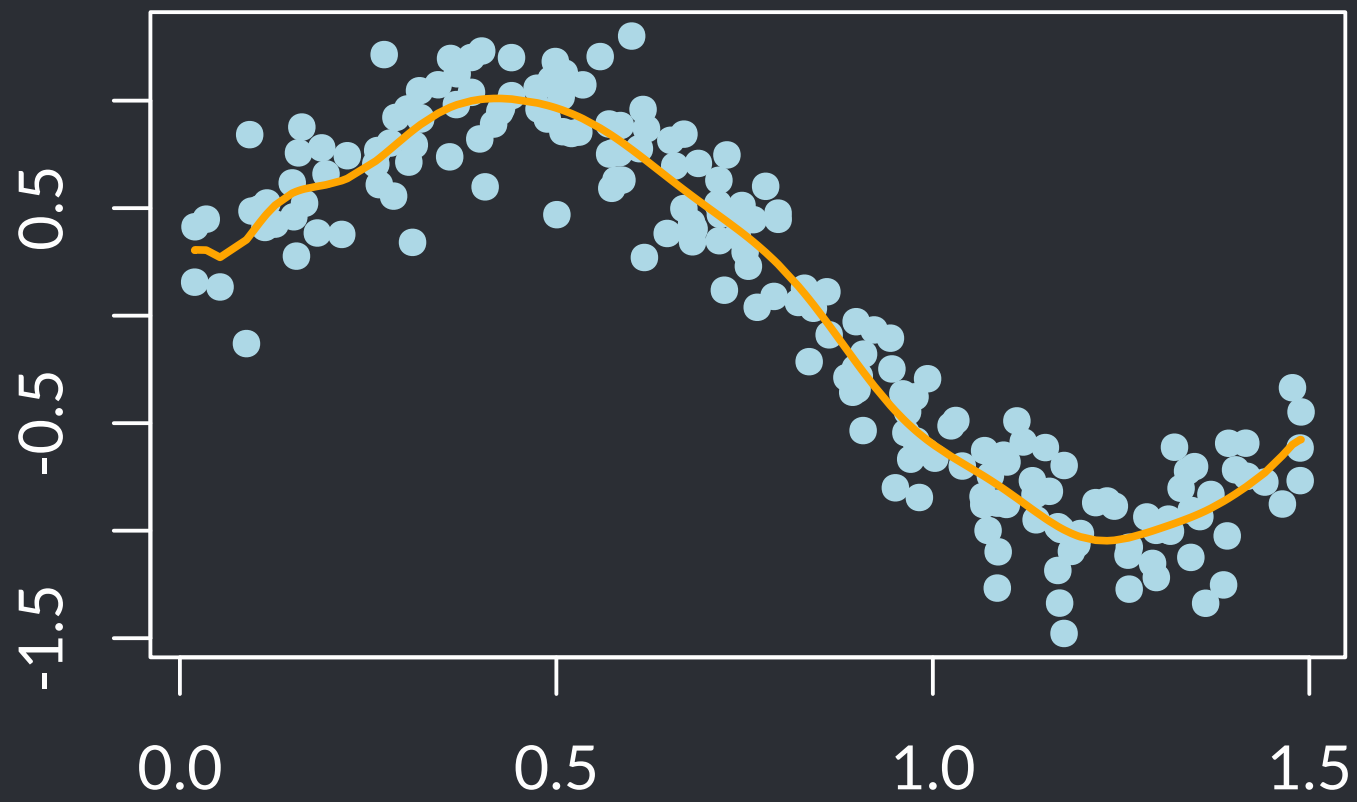
average absolute prediction error = 0.1565

Degree 20 polynomial



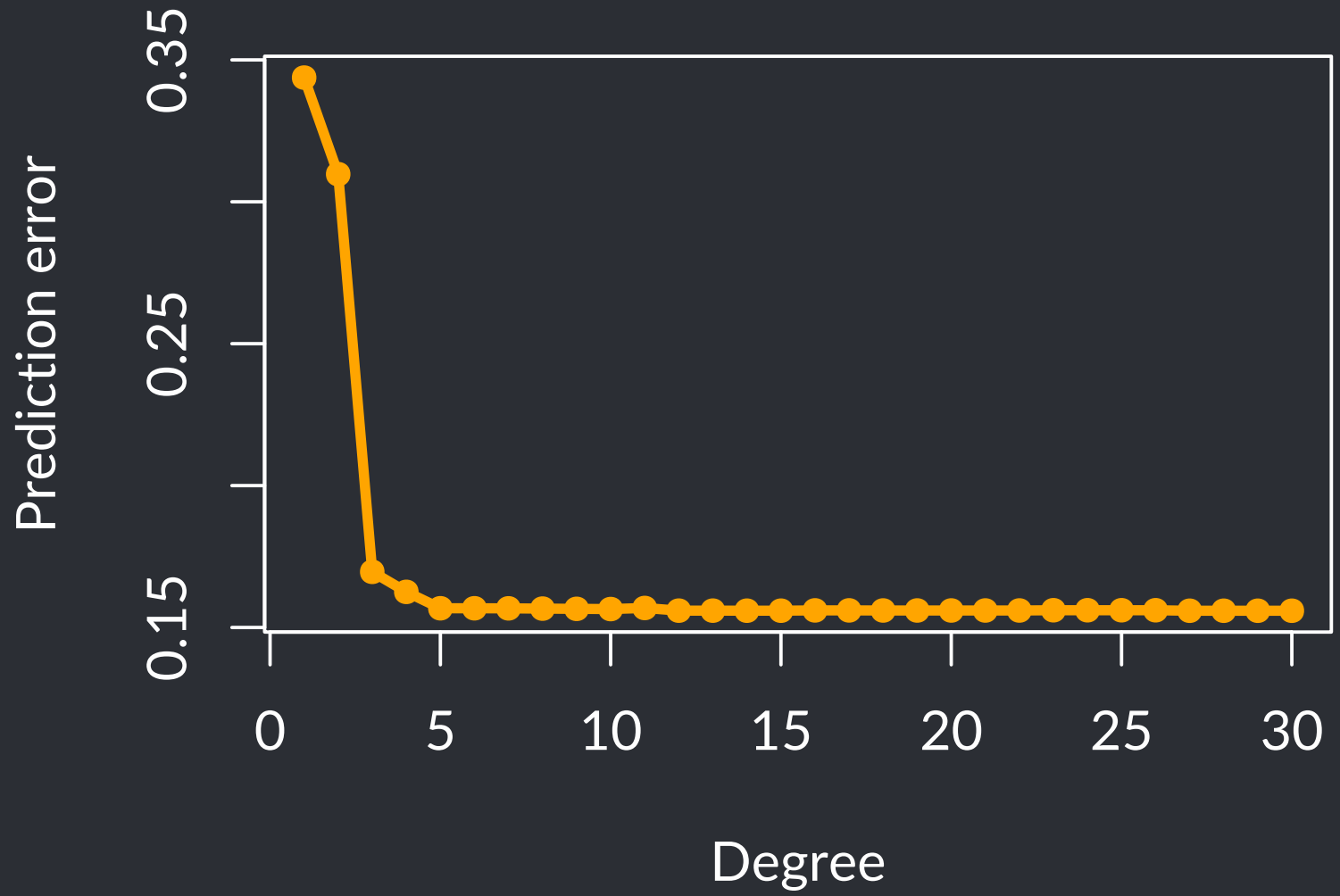
average absolute prediction error = 0.156

Degree 30 polynomial



average absolute prediction error = 0.1559

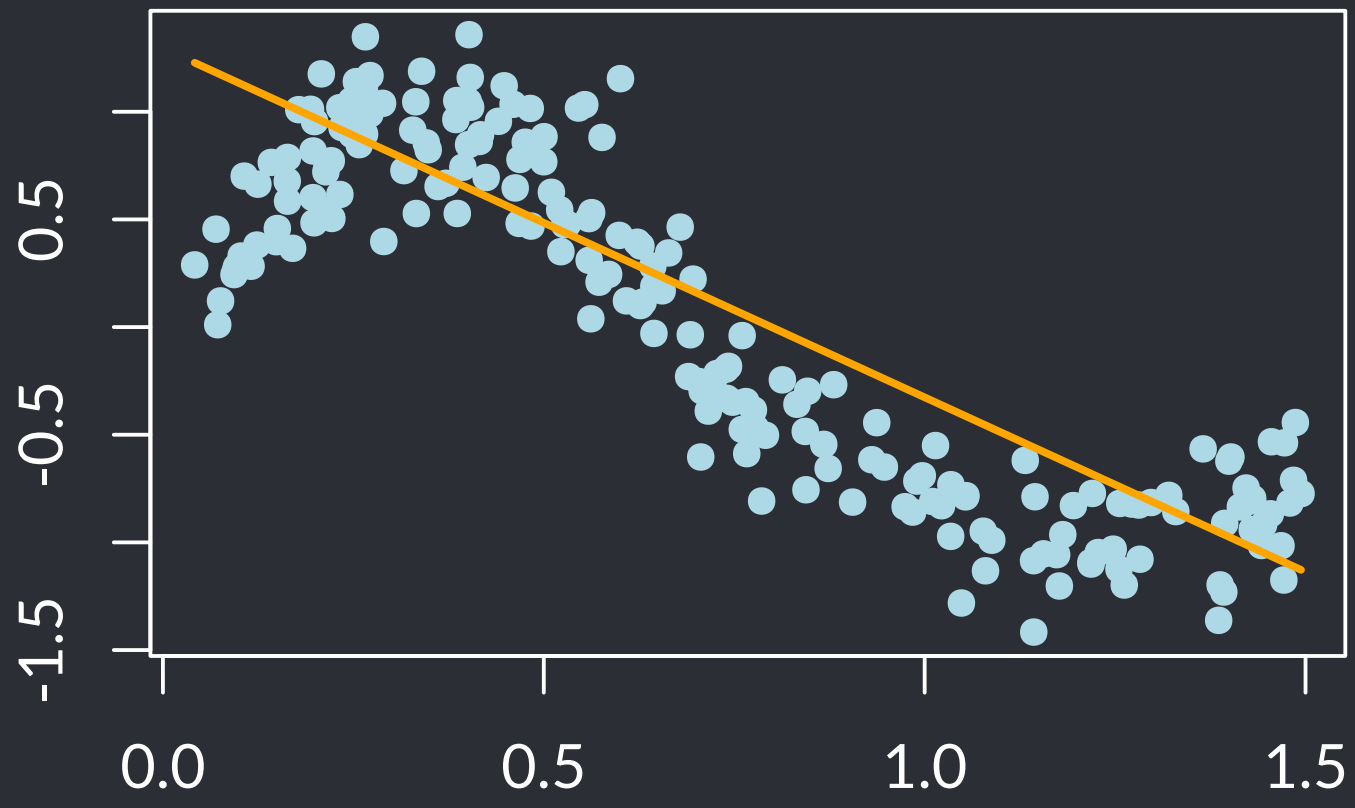
As we fit increasingly complex polynomials, the average prediction error decreases:



Parsimony

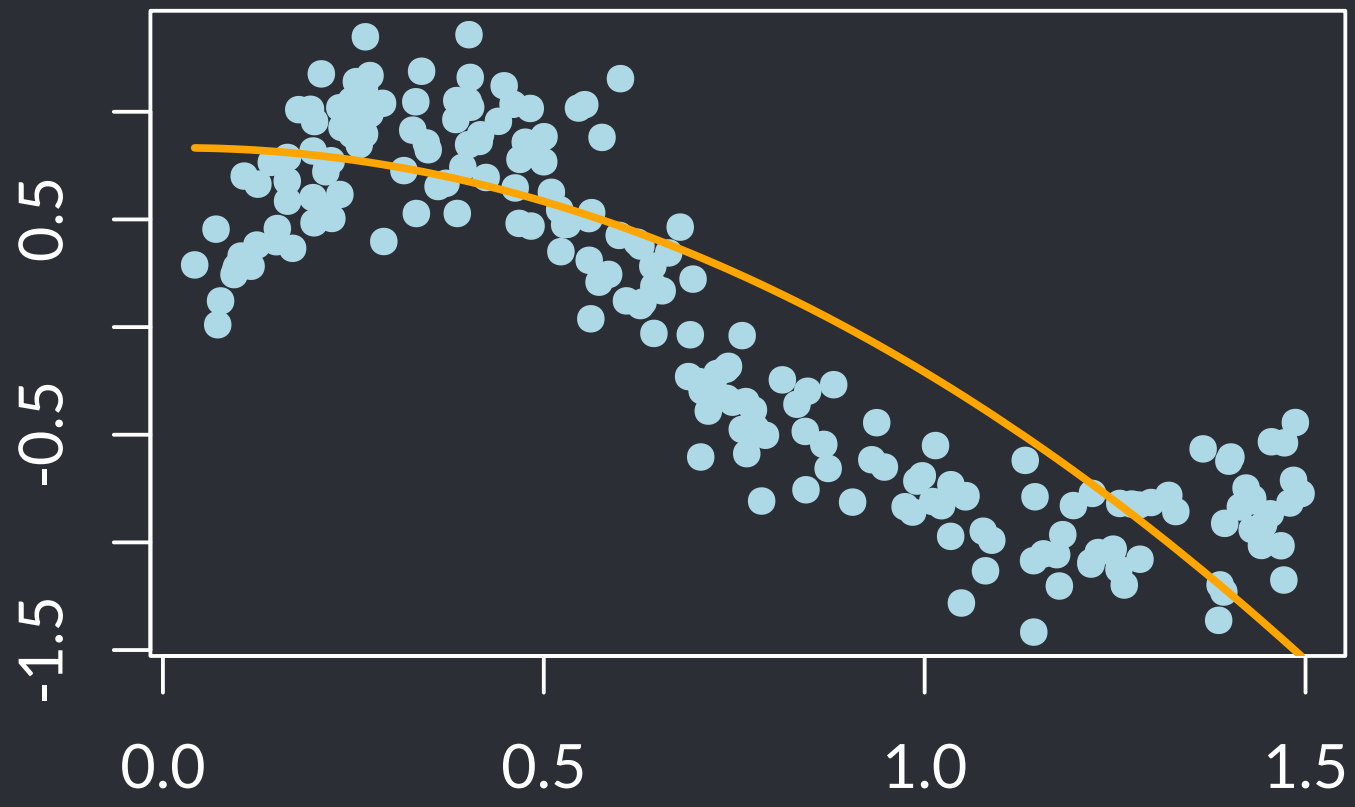
- We've talked a lot about the need to select models that are parsimonious, i.e., those that strike a good balance between fitting well and being simple.
- Besides being easier to communicate, there's another reason to prefer simpler models—they tend to generalize better to new data.
- Let's see how these models perform on a new sample from the same population!

Degree 1 polynomial



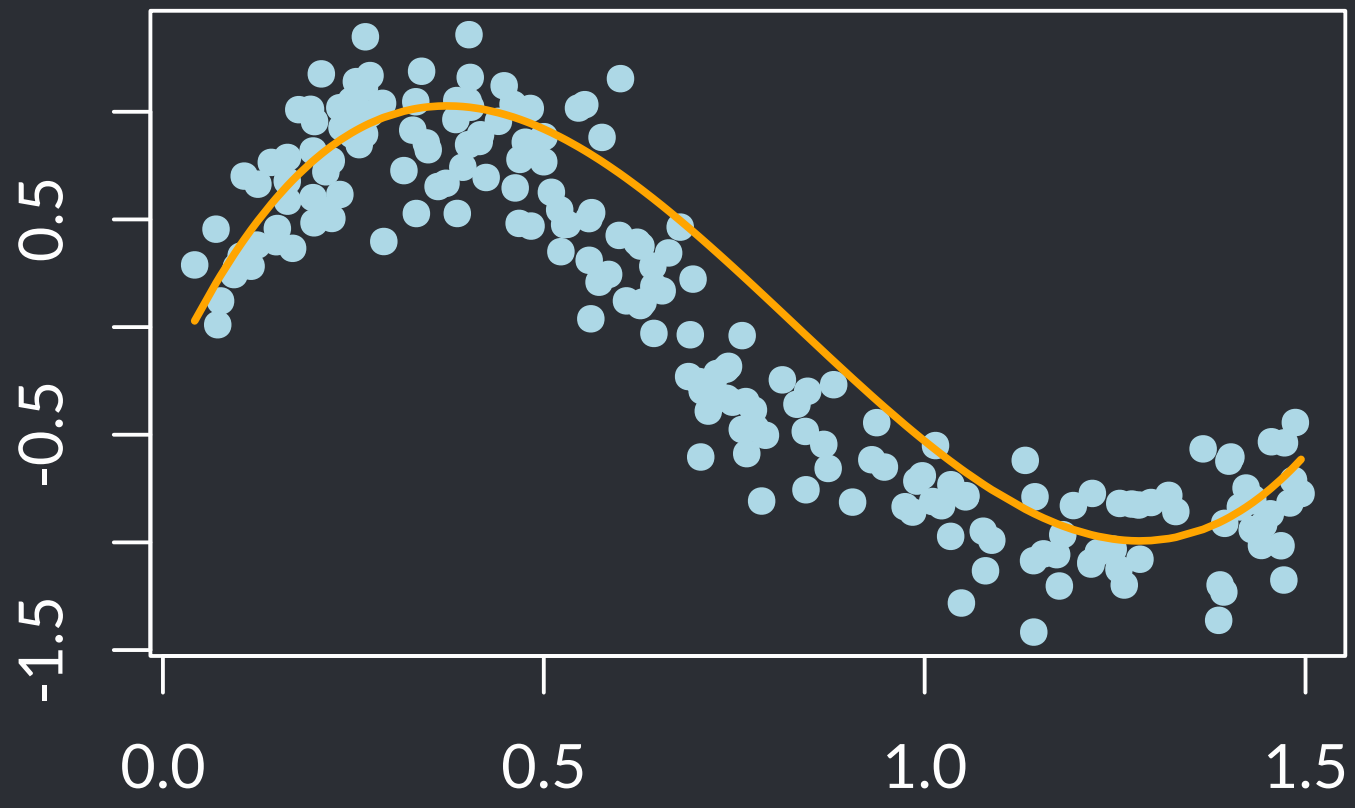
average absolute prediction error = 0.3339 (was 0.3438 in the original sample)

Degree 2 polynomial



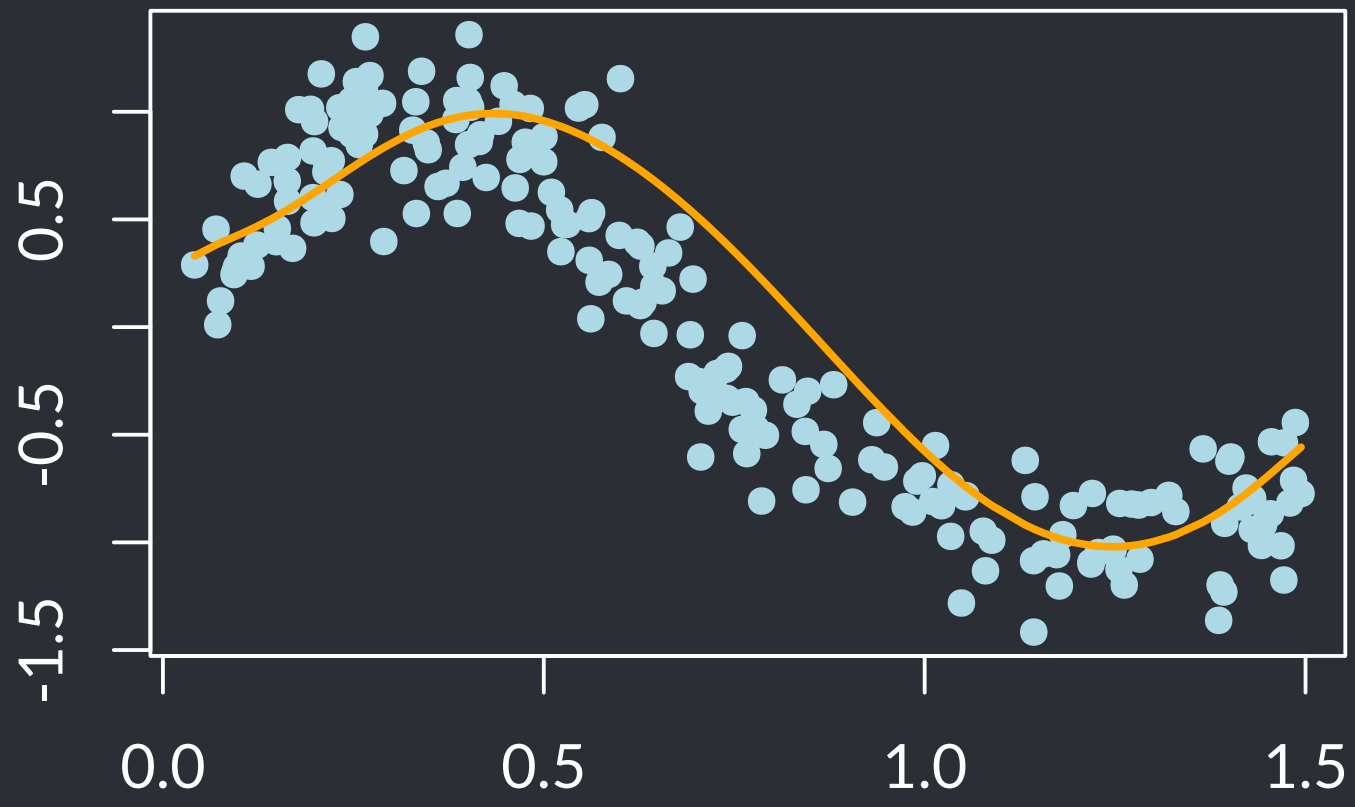
average absolute prediction error = 0.3688 (was 0.3097 in the original sample)

Degree 3 polynomial



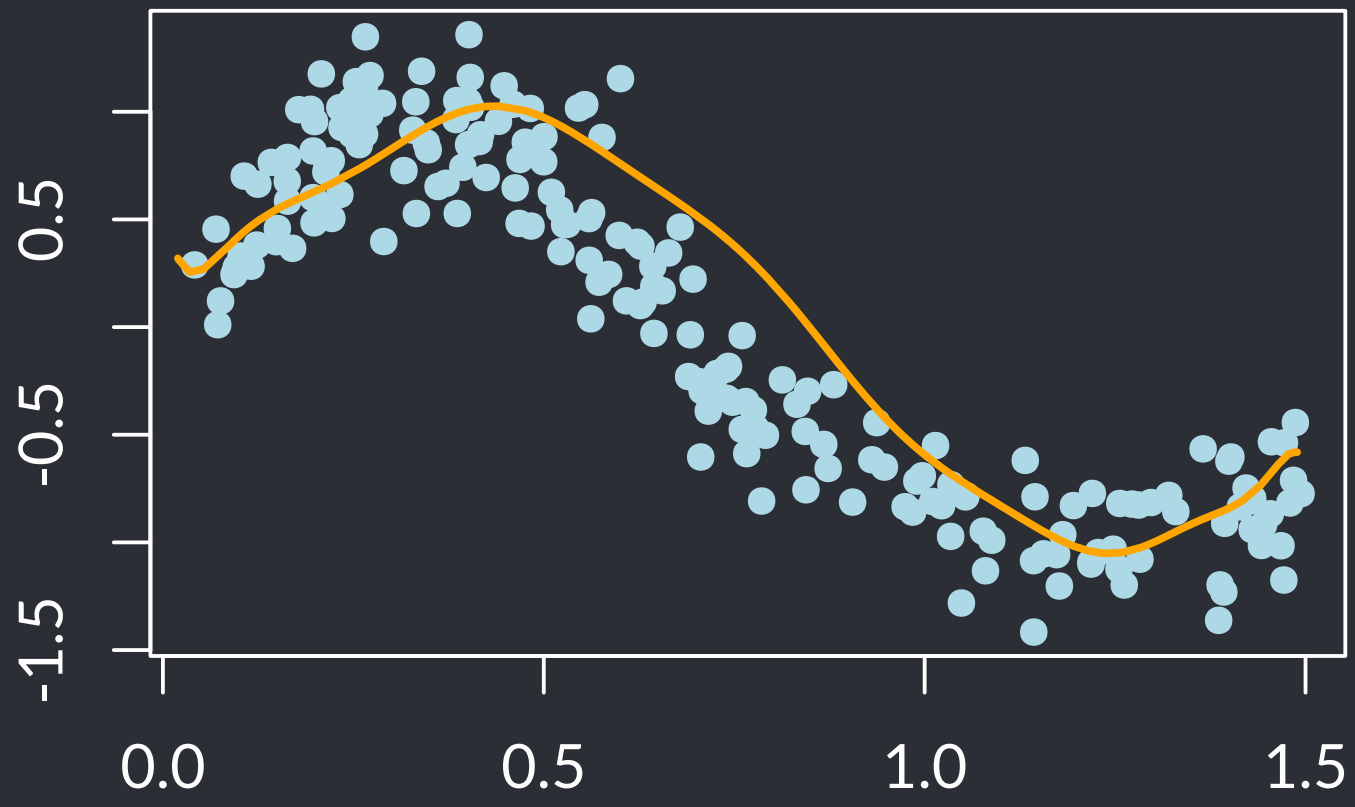
average absolute prediction error = 0.2718 (was 0.1696 in the original sample)

Degree 10 polynomial



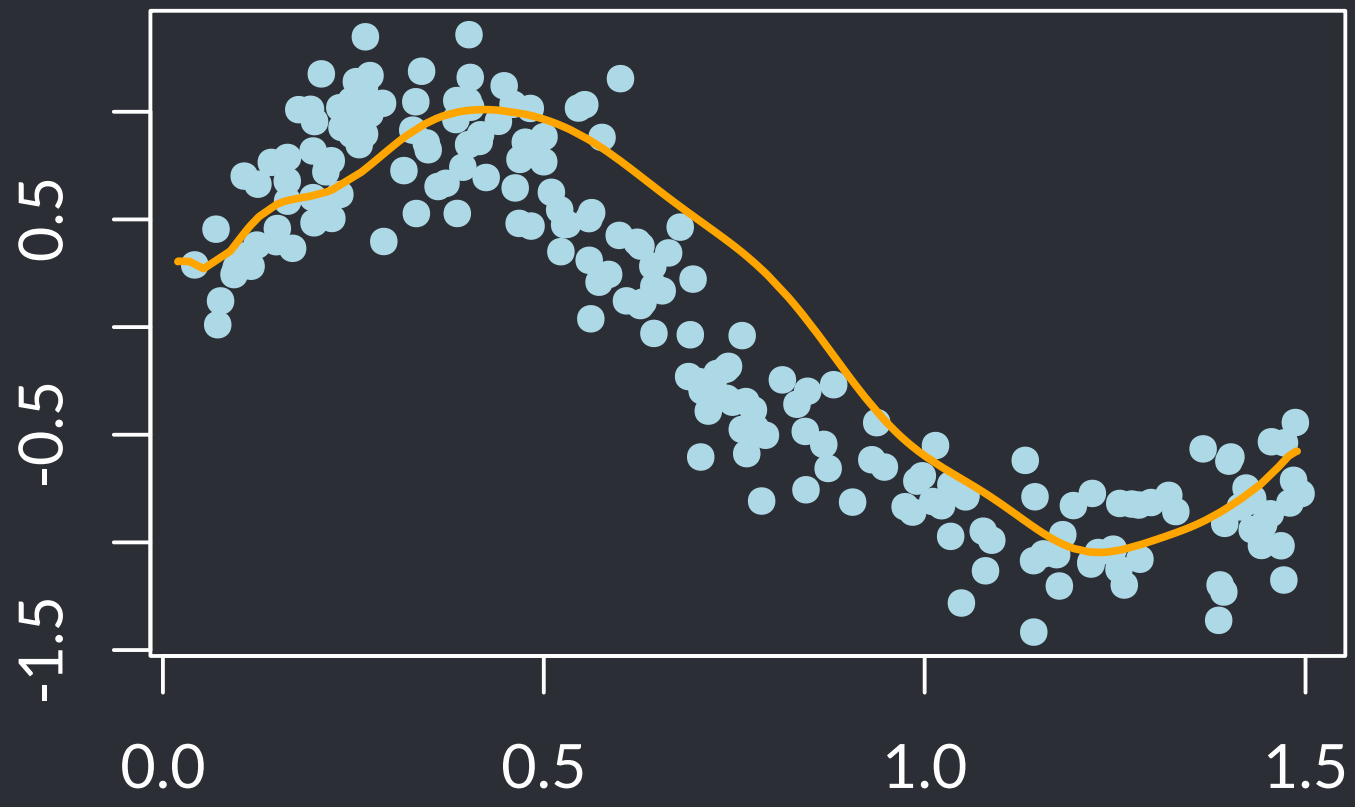
average absolute prediction error = 0.2927 (was 0.1565 in the original sample)

Degree 20 polynomial



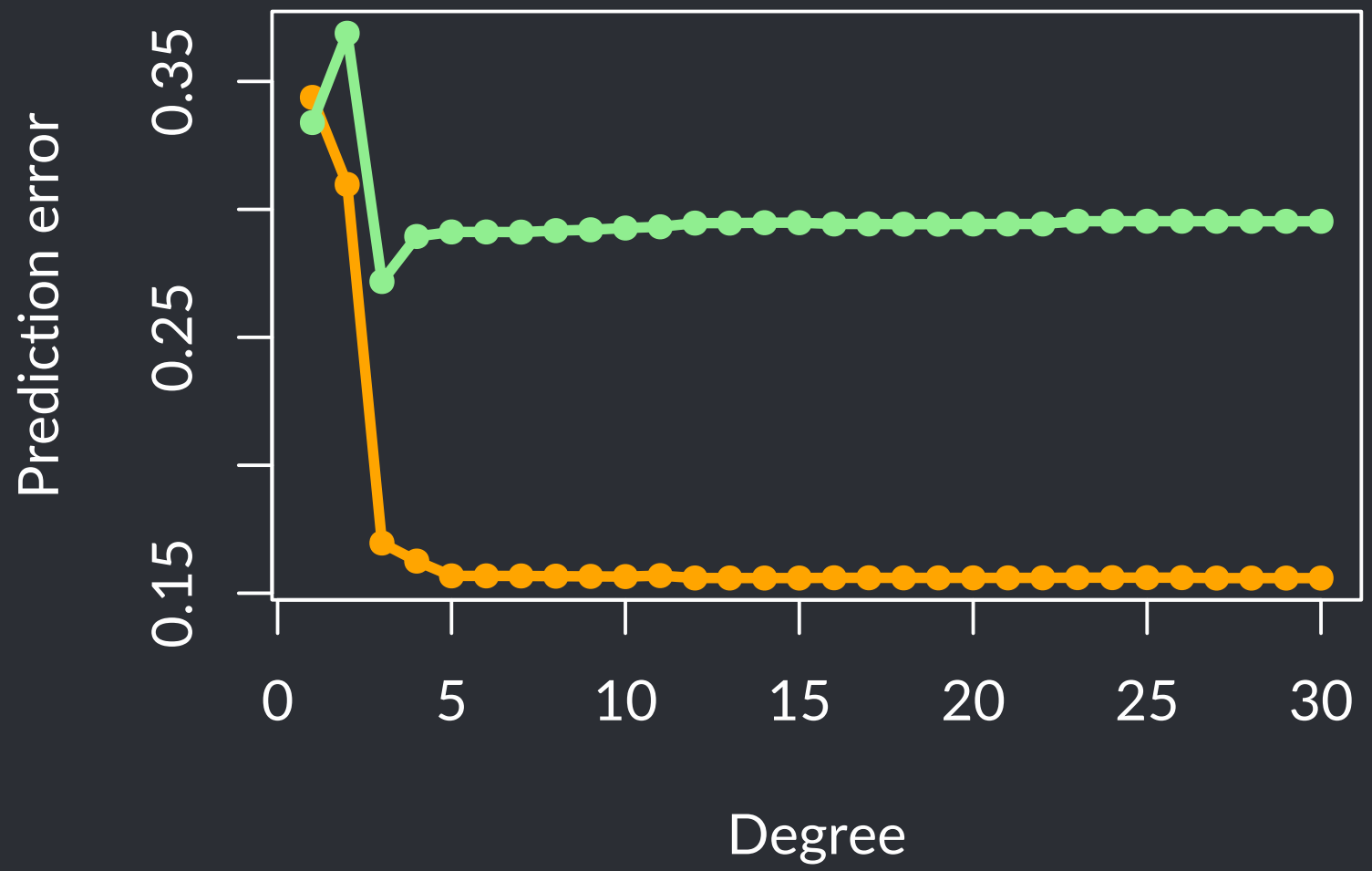
average absolute prediction error = 0.2927 (was 0.156 in the original sample)

Degree 30 polynomial



average absolute prediction error = 0.2927 (was 0.1559 in the original sample)

The increasingly complex polynomials do not perform as well on the new data as the simple polynomials, because they had overfit the idiosyncrasies of the original data.



Overfitting in multiple regression

- A similar phenomenon occurs when you have many variables to choose from when building a regression model.

Overfitting in multiple regression

- A similar phenomenon occurs when you have many variables to choose from when building a regression model.
- By including many variables in your model, you can get R^2 to increase (or prediction error to decrease) on the data used to build the model.

Overfitting in multiple regression

- A similar phenomenon occurs when you have many variables to choose from when building a regression model.
- By including many variables in your model, you can get R^2 to increase (or prediction error to decrease) on the data used to build the model.
- But the more complex models are more likely to be overfitting the data, and won't generalize as well as simple models.

Overfitting in multiple regression

- A similar phenomenon occurs when you have many variables to choose from when building a regression model.
- By including many variables in your model, you can get R^2 to increase (or prediction error to decrease) on the data used to build the model.
- But the more complex models are more likely to be overfitting the data, and won't generalize as well as simple models.
- In general, a model's performance on the data used to build the model will usually be stronger than its performance on new data.

1. Overfitting

2. Using *p*-values responsibly to avoid overfitting and *p*-hacking

3. Training and test sets

Two phases of analysis

- Think of any analysis you do as having two phases, rather than one:
  - The exploratory phase where you try out different conditions, different subgroups, different measures of success, etc. until you find something that you *think* works.
  - The confirmatory phase where you try to replicate your results in a new sample.

The exploratory phase in regression analysis

- Try looking at a large number of variables.
- Try looking at different subsets—e.g. only women, or only men; only large companies, or only small companies; etc.
- Use low *p*-values as a guide to what *might* generalize to the larger population, but take them with a grain (cannister?!) of salt.

The confirmatory phase in regression analysis

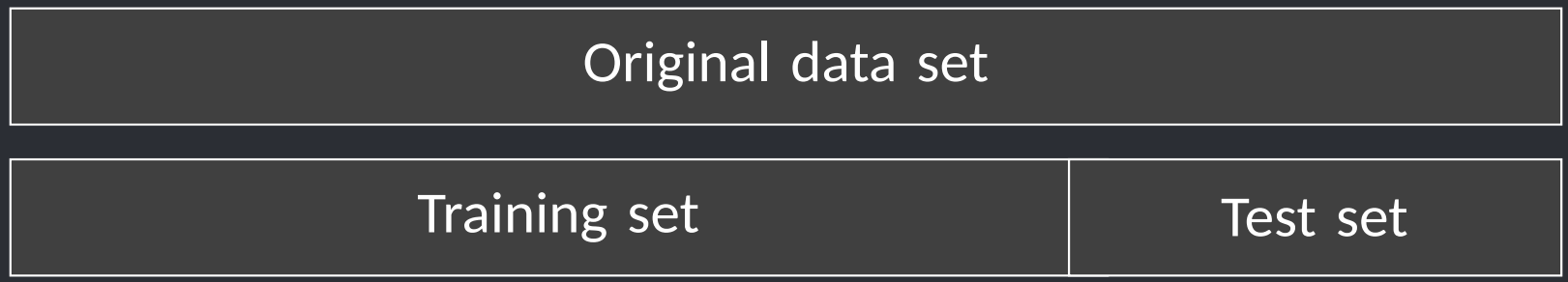
- Using what you think is the best model from the exploratory phase, see if your results generalize to a completely new sample.
- Now you can trust that your *p*-values are not misleading, since you are only running a single test on this sample!

1. Overfitting

2. Using *p*-values responsibly to avoid overfitting and *p*-hacking

3. Training and test sets

Review of training and test sets



- Split the data into a training set and a test set (a typical split is 70% training set / 30% test set).
- We use the training set to build the model, and then evaluate the quality of the model on how well it predicts Y in the test set.

The housing data set

Let's look at different models to predict housing prices, with different sets of predictors.

Let's split our data into a training and test set with a rough 70/30 split. ($N = 1728$, and 30% of 1728 is approximately 518.)

```
# Select which row numbers will correspond to test cases
test.cases <- sample(1:1728, 518)

# Build the test set from those row numbers
test.set <- houses[test.cases,]

# Remaining row numbers will correspond to training cases
training.cases <- setdiff(1:1728, test.cases)

# Build the training set from those row numbers
training.set <- houses[training.cases,]
```

Model-building strategy

- At first, the temptation is to build models on the training set, test using the test set, and repeat.

Model-building strategy

- At first, the temptation is to build models on the training set, test using the test set, and repeat.
- But this would lead to the same overfitting and *p*-hacking issues as before!

Model-building strategy

- At first, the temptation is to build models on the training set, test using the test set, and repeat.
- But this would lead to the same overfitting and *p*-hacking issues as before!
- Instead, set aside the test set and don't look at it until you have a final model: what you think is the most parsimonious model.

Model-building strategy

- At first, the temptation is to build models on the training set, test using the test set, and repeat.
- But this would lead to the same overfitting and *p*-hacking issues as before!
- Instead, set aside the test set and don't look at it until you have a final model: what you think is the most parsimonious model.
- Evaluate model performance in the test set as a way to get a fair measurement of how well your model will perform on new data.

Evaluating training and test error

Let's start with a simple model that uses living area only:

```
model <- lm(Price ~ Living.Area, data=training.set)
```

The training set average error is:

```
mean(abs(resid(model)))
```

```
[1] 47685.76
```

The test set average error comes from manually computing the prediction error for each case in the test set:

```
price.hat <- predict(model, test.set)
mean(abs(test.set$Price - price.hat))
```

```
[1] 47983.22
```

Evaluating training and test error

- Similarly, we can compare the R^2 from the training set to the R^2 that we would get by predicting prices for cases in the test set.

Evaluating training and test error

- Similarly, we can compare the R^2 from the training set to the R^2 that we would get by predicting prices for cases in the test set.
- Recall that $R^2 = \text{cor}(Y, \hat{Y})^2$; we can simulate what R^2 would be in the test set by calculating this in the test set:

```
cor(test.set$Price, price.hat)^2
```

```
[1] 0.5162538
```

Evaluating training and test error

- Similarly, we can compare the R^2 from the training set to the R^2 that we would get by predicting prices for cases in the test set.
- Recall that $R^2 = \text{cor}(Y, \hat{Y})^2$; we can simulate what R^2 would be in the test set by calculating this in the test set:

```
cor(test.set$Price, price.hat)^2
```

```
[1] 0.5162538
```

- Compare this to what R^2 is in the training set:

```
summary(model)$r.squared
```

```
[1] 0.5040246
```

Using training and test sets in logistic regression

- The logic of training and test sets is identical for logistic regression.
- If we build a logistic model to predict something, using a test set for evaluation will give us a more realistic estimate of prediction accuracy on new data.
- Let's try this out on a data set of passengers from the *Titanic*.

The data set

The *Titanic* data set has data on 756 passengers on the *Titanic*; we'll predict the categorical variable survival from age of the passenger. Let's segment the data into training and test sets, as before:

```
# Select which row numbers will correspond to test cases
test.cases <- sample(1:756, 227)

# Build the test set from those row numbers
test.set <- titanic[test.cases,]

# Remaining row numbers will correspond to training cases
training.cases <- setdiff(1:756, test.cases)

# Build the training set from those row numbers
training.set <- titanic[training.cases,]
```

Let’s build the model:

```
model <- glm(Survived ~ Age, data=training.set, family=binomial)
```

Let's build the model:

```
model <- glm(Survived ~ Age, data=training.set, family=binomial)
```

Then we can evaluate prediction accuracy as usual:

```
predicted <- (predict(model, type="response") >= 0.5)
actual <- (training.set$Survived == 1)
sum(predicted == actual) / nrow(training.set)
```

```
[1] 0.5897921
```

Let's build the model:

```
model <- glm(Survived ~ Age, data=training.set, family=binomial)
```

Then we can evaluate prediction accuracy as usual:

```
predicted <- (predict(model, type="response") >= 0.5)
actual <- (training.set$Survived == 1)
sum(predicted == actual) / nrow(training.set)
```

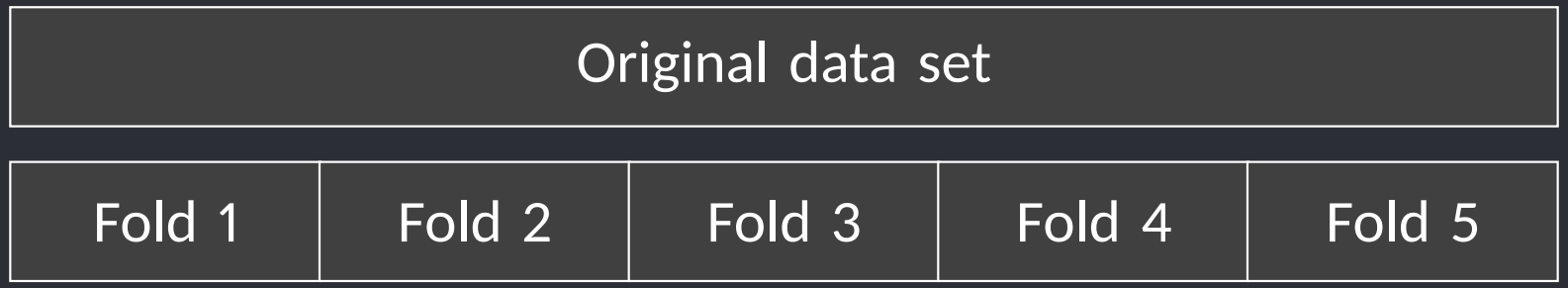
```
[1] 0.5897921
```

To evaluate prediction accuracy on the training set, we use the same model, but apply it to the test set:

```
predicted <- (predict(model, test.set, type="response") >= 0.5)
actual <- (test.set$Survived == 1)
sum(predicted == actual) / nrow(test.set)
```

```
[1] 0.5770925
```

Cross-validation



- Split the data into k "folds" (here $k = 5$).
- For each fold, use that fold as a test set and the other folds together as a training set.
- Average the prediction accuracy across all k folds as your best estimate of prediction accuracy.
- Cross-validation reduces the impact of the random chance from your selection of training and test sets.