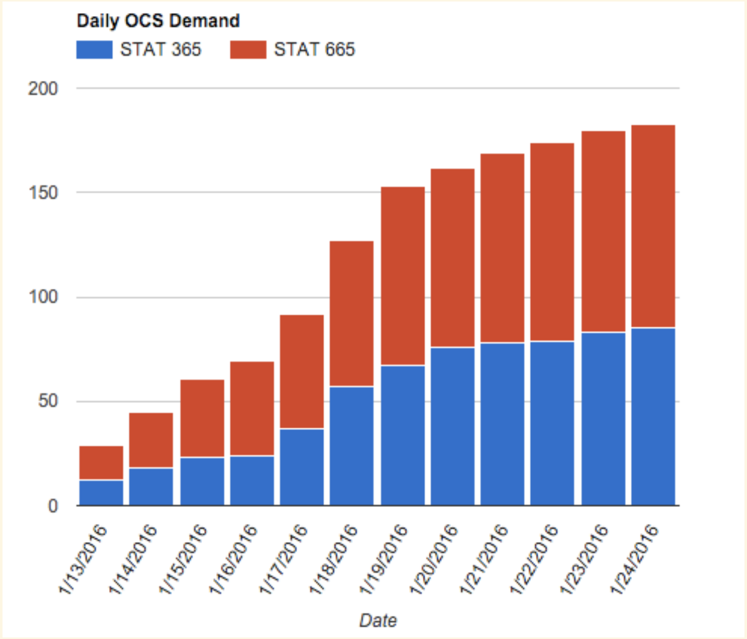
Lecture 03
Linear classification methods II

25 January 2016

Taylor B. Arnold
Yale Statistics
STAT 365/665

- Office hours
  - Taylor Arnold – Mondays, 13:00 - 14:15, HH 24, Office 203 (by appointment)
  - Yu Lu – Tuesdays, 10:00-12:00, HH 24, Library
  - Jason Klusowski – Thursdays, 19:00-20:30, HH 24
- Problem set #2 - Posted, on ClassesV2, updated datasets (as of 2015-01-23)
- Enrollment...


Daily OCS Demand
STAT 365
STAT 665
200
150
100
50
0
1/13/2016
1/14/2016
1/15/2016
1/16/2016
1/17/2016
1/18/2016
1/19/2016
1/20/2016
1/21/2016
1/22/2016
1/23/2016
1/24/2016
Date

Last class we considered solutions to the 1-dimensional non-parametric regression problem. Specifically, we considered observing n pairs (x_i, y_i) such that:

$$y_i = g(x_i) + \epsilon_i$$

For an unknown function g and random variable ϵ_i, where the mean of the random variable is zero.

The goal was to estimate the function g.

The specific methods we used to do this were:

1. k-nearest neighbors (knn)
2. kernel smoothing (Nadaraya-Watson estimator)
3. linear regression (OLS)
4. local regression (LOESS)

Linear smoothers

How are these four techniques all related?

Linear smoothers

How are these four techniques all related?

All of them can be written as:

$$\begin{aligned}\widehat{y}_{new} &= \widehat{g}(x_{new}) \\ &= \sum_i w_i y_i\end{aligned}$$

Where the weights w_i depend only on the values x_i (and x_{new}) and not on the values of y_i.

Linear smoothers

How are these four techniques all related?

All of them can be written as:

$$\begin{aligned}\widehat{y}_{new} &= \widehat{g}(x_{new}) \\ &= \sum_i w_i y_i\end{aligned}$$

Where the weights w_i depend only on the values x_i (and x_{new}) and not on the values of y_i.

Anything that can be written in this form is called a linear smoother.

Linear smoothers, cont.

For example, in the case of knn, the weights are $\frac{1}{k}$ for the k closest values and 0 for the rest of them.

Linear smoothers, cont.

For example, in the case of knn, the weights are $\frac{1}{k}$ for the k closest values and 0 for the rest of them.

We have already seen the weights for the kernel smoother by its definition.

Linear smoothers, cont.

For example, in the case of knn, the weights are $\frac{1}{k}$ for the k closest values and 0 for the rest of them.

We have already seen the weights for the kernel smoother by its definition.

The weights for ordinary least squares are given by the matrix product (X is the data matrix from basis expansion of the inputs):

$$w = X_{new}(X^t X)^{-1} X^t$$

Linear smoothers, cont.

For example, in the case of knn, the weights are $\frac{1}{k}$ for the k closest values and 0 for the rest of them.

We have already seen the weights for the kernel smoother by its definition.

The weights for ordinary least squares are given by the matrix product (X is the data matrix from basis expansion of the inputs):

$$w = X_{new}(X^t X)^{-1} X^t$$

And LOESS is simply a sample weighted variant of the least squares:

$$w = X_{new}(X^t D X)^{-1} X^t D$$

For a diagonal matrix of weights $D = \text{diag}\left(\phi(||x_{new} - x_i||_2)\right)$.

Linear smoothers, cont.

There is a substantial amount of theory regarding the class of linear smoothers.

If you are interested in more, the best first reference is the following text (you can download a free pdf):

> *Cosma Rohilla Shalizi. Advanced Data Analysis from an Elementary Point of View. Book in preparation.* `http://www.stat.cmu.edu/~cshalizi/ADAfaEPoV/`.

Computational issues

I have not yet talked about how these algorithms should be implemented. Problem set #1 asks you to write your own implementation of knn and kernel smoothing; brute force is fine for the assignment, but can we do better?

Computational issues, cont.

If we want to classify m new data points, and have n observations in the original dataset, what is the computational complexity of the brute force knn or kernel smoother?

Computational issues, cont.

If we want to classify m new data points, and have n observations in the original dataset, what is the computational complexity of the brute force knn or kernel smoother?

We need to construct the entire m by n distance matrix, and so this takes a total of $m \cdot n$ operations. For a small m, this is actually quite satisfactory. For a larger value of m, refitting on the entire training set for example, this can become prohibitive for large sample sizes.

Computational issues, cont.

Consider first sorting the inputs:

$$x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$$

This can be done in $O(n \cdot log(n))$ time.

Computational issues, cont.

Consider first sorting the inputs:

$$x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$$

This can be done in $O(n \cdot log(n))$ time.

Using binary search, we can insert a new observation x_{new} into this sorted sequence in $O(log(n))$ operations:

$$x_r \leq x_{new} \leq x_{r+1}$$

Computational issues, cont.

Consider first sorting the inputs:

$$x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$$

This can be done in $O(n \cdot log(n))$ time.

Using binary search, we can insert a new observation x_{new} into this sorted sequence in $O(log(n))$ operations:

$$x_r \leq x_{new} \leq x_{r+1}$$

For knn, we can then look at only the k neighbors less than x_{new} and the k neighbors greater than x_{new}. The k nearest neighbors are guaranteed to be in this set. This only takes $2k$ time.

Computational issues, cont.

Consider first sorting the inputs:

$$x_1 \leq x_2 \leq \cdots \leq x_{n-1} \leq x_n$$

This can be done in $O(n \cdot log(n))$ time.

Using binary search, we can insert a new observation x_{new} into this sorted sequence in $O(log(n))$ operations:

$$x_r \leq x_{new} \leq x_{r+1}$$

For knn, we can then look at only the k neighbors less than x_{new} and the k neighbors greater than x_{new}. The k nearest neighbors are guaranteed to be in this set. This only takes $2k$ time.

This implementation takes only $O(mk \cdot \log(n))$ operations to fit new observations.

Computational issues, cont.

Now, consider the following intervals:

$$B_q = [x_{min} + h \cdot (q-1), \quad x_{min} + h \cdot q)$$

For some fixed bandwidth h. There should be $range(x)/h$ such buckets, a constant we will define as N.

Computational issues, cont.

Now, consider the following intervals:

$$B_q = [x_{min} + h \cdot (q-1), \quad x_{min} + h \cdot q)$$

For some fixed bandwidth h. There should be $range(x)/h$ such buckets, a constant we will define as N.

Partitioning the input variables x_i into these buckets takes only n operations.

Computational issues, cont.

Now, consider the following intervals:

$$B_q = [x_{min} + h \cdot (q-1), \quad x_{min} + h \cdot q)$$

For some fixed bandwidth h. There should be $range(x)/h$ such buckets, a constant we will define as N.

Partitioning the input variables x_i into these buckets takes only n operations.

If we use kernel smoothing truncated at a distance $h/2$ away from the mean, we can estimate $\widehat{y_{new}}$ for a new input by only looking at points inside 2 of the buckets B_q. How long this takes depends on how evenly distributed the data are across the range of x.

Computational issues, cont.

Now, consider the following intervals:

$$B_q = [x_{min} + h \cdot (q-1), \quad x_{min} + h \cdot q)$$

For some fixed bandwidth h. There should be $range(x)/h$ such buckets, a constant we will define as N.

Partitioning the input variables x_i into these buckets takes only n operations.

If we use kernel smoothing truncated at a distance $h/2$ away from the mean, we can estimate $\widehat{y_{new}}$ for a new input by only looking at points inside 2 of the buckets B_q. How long this takes depends on how evenly distributed the data are across the range of x.

If the inputs are evenly spread out over the input space, we can fit kernel smoothing on m new inputs in $O(n + mn/N)$ time.

Computational issues, cont.

In one dimension these algorithms are fairly straightforward. We will continue to discuss how to implement these when the number of dimensions is larger. Sorting obviously does not work directly; bucketing *can* for a reasonable number of dimensions but becomes ineffective for large dimensional spaces.

Tuning parameters

Last class we spent a while looking at how these four estimators vary with respect to their tuning parameters.

How do we actually choose the value of the tuning parameter to use a prediction system?

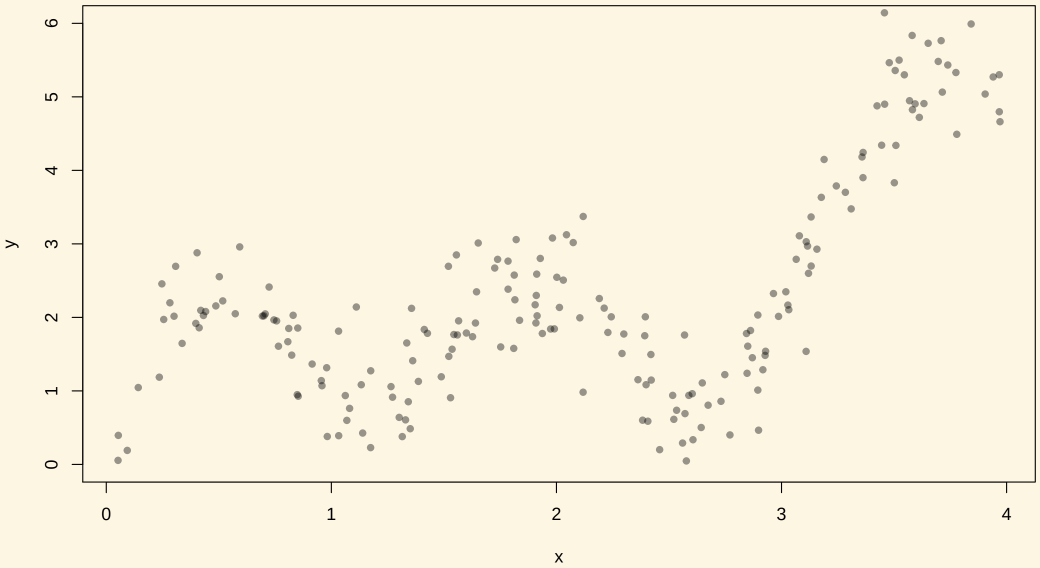
Tuning parameters, cont.

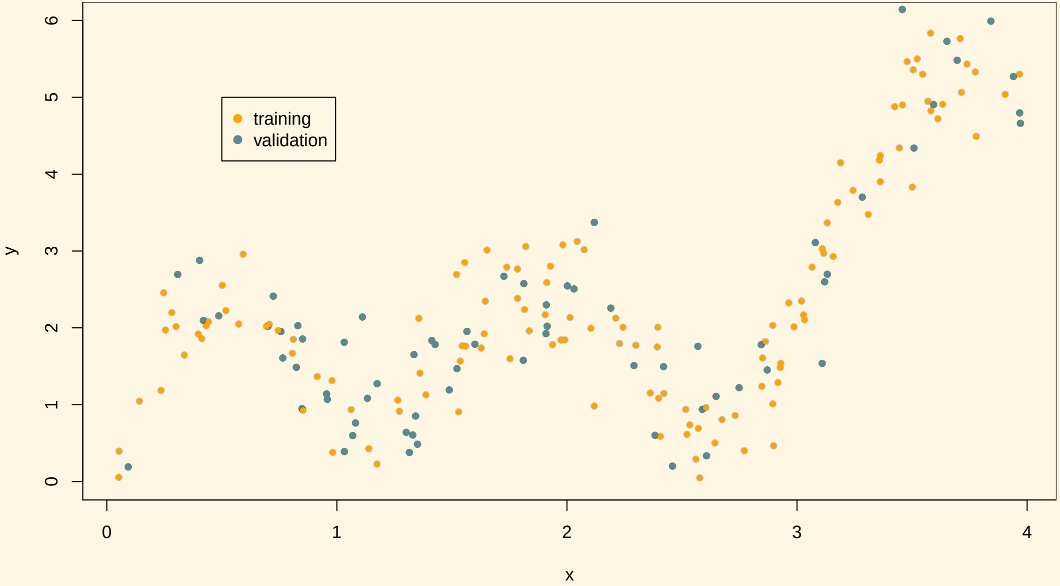
There is a lot of theory that establishes asymptotic formulas for the form of some tuning parameters. For example, the bandwidth in kernel smoothing should be of the form $O(n^{-1/5})$.

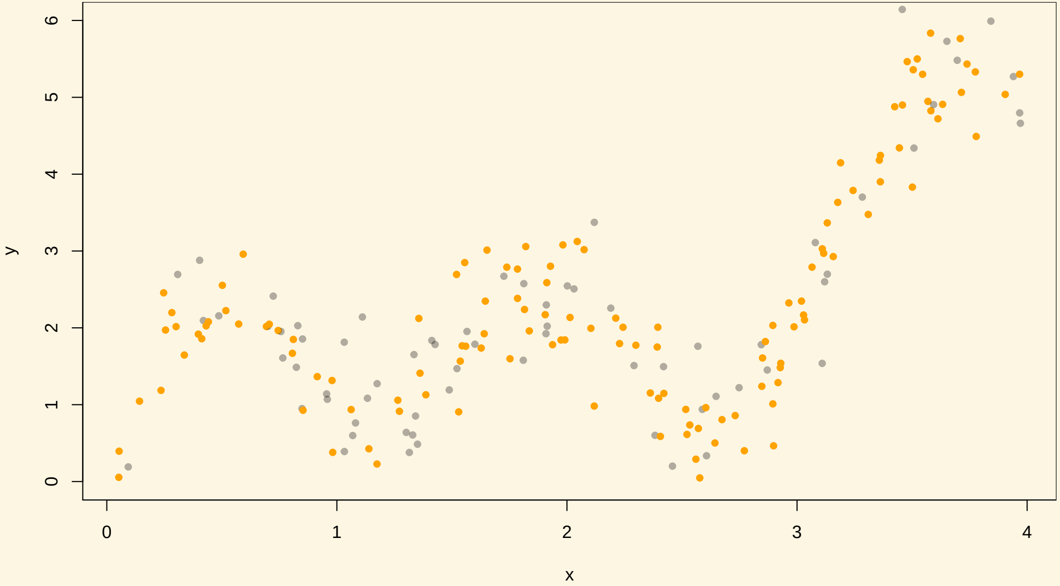
Tuning parameters, cont.

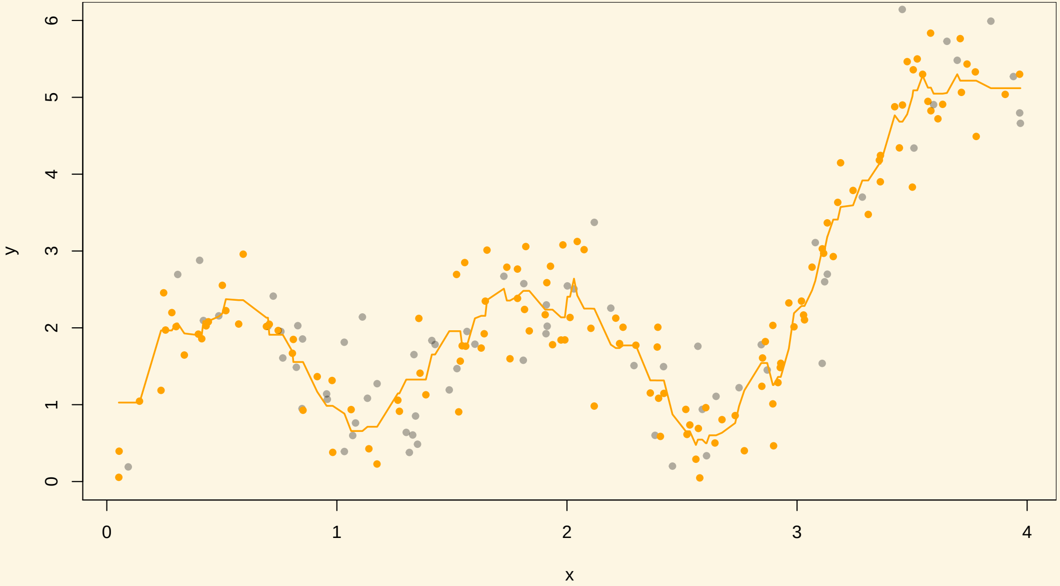
There is a lot of theory that establishes asymptotic formulas for the form of some tuning parameters. For example, the bandwidth in kernel smoothing should be of the form $O(n^{-1/5})$.

Unfortunately, in almost every case I know of these theoretical results yield poor results when directly applied to finite samples. We need to instead estimate the value of the tuning parameter from the data itself.

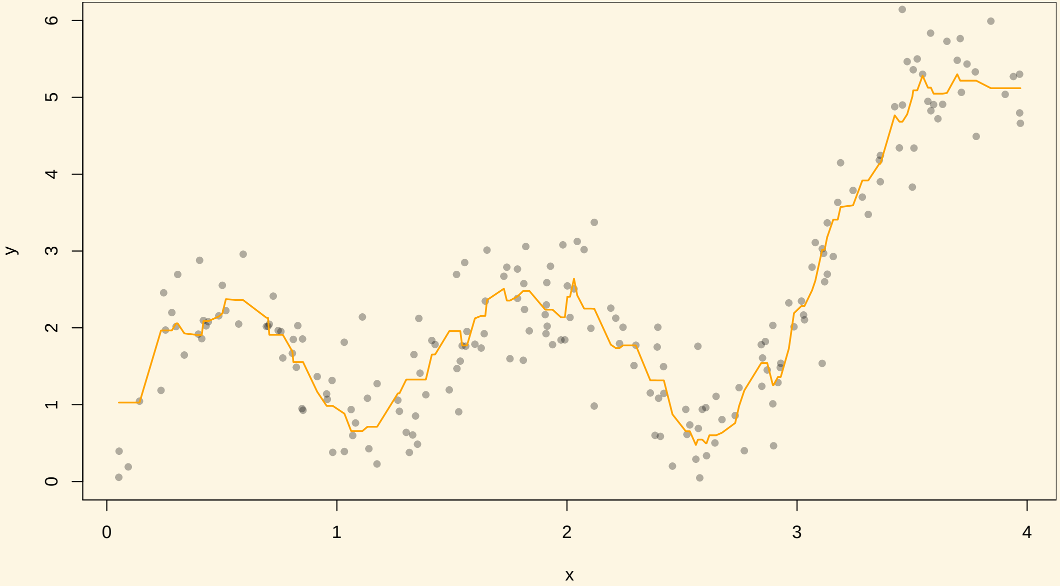

y
0
1
2
3
4
5
6
x
0
1
2
3
4


training
validation
y
x
0
1
2
3
4
5
6
0
1
2
3
4

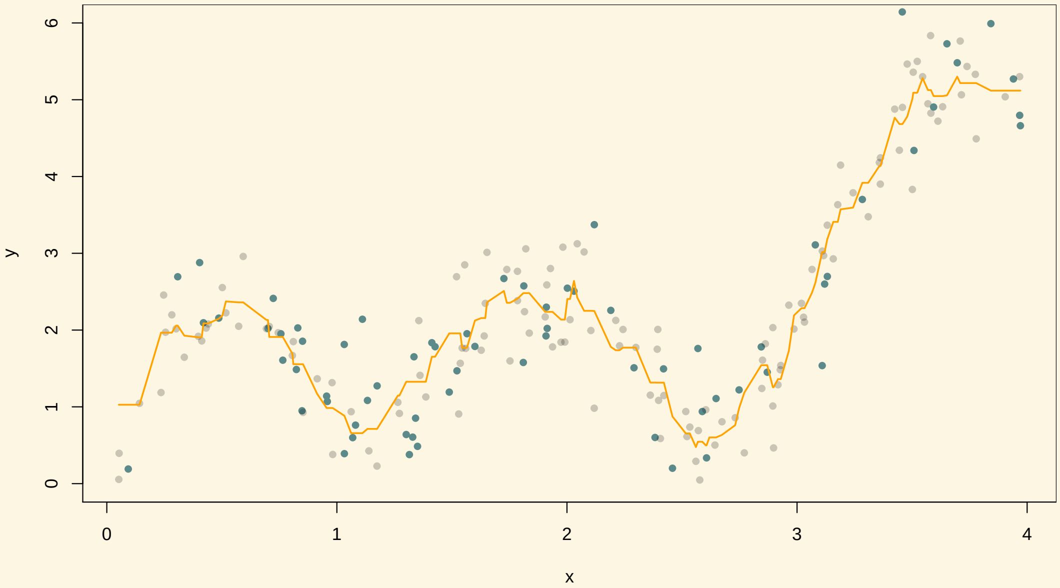

y
x
0
1
2
3
4
5
6
0
1
2
3
4

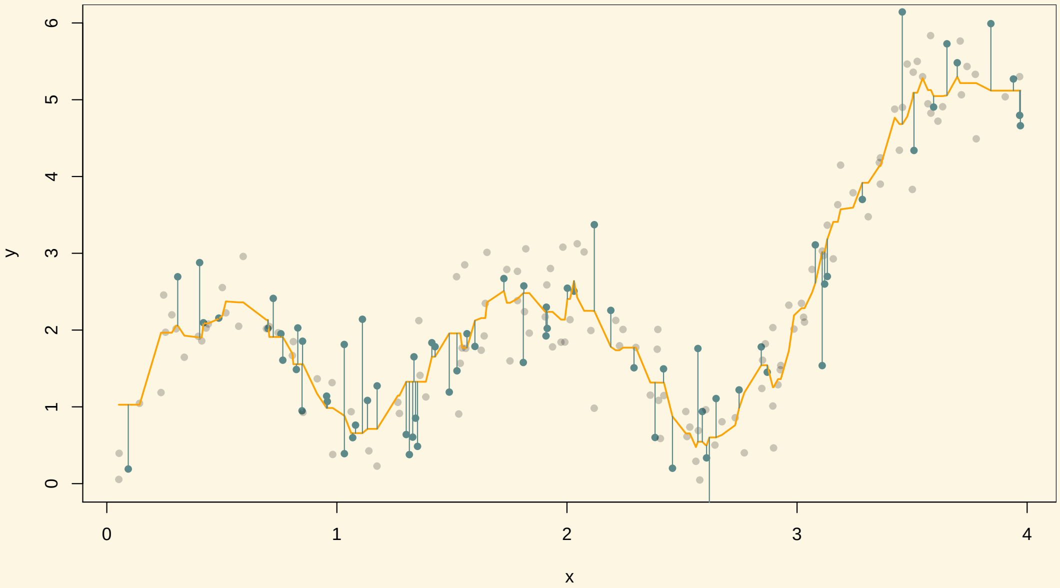


y
x
0
1
2
3
4
5
6



y
0
1
2
3
4
5
6
x
0
1
2
3
4


y
0
1
2
3
4
5
6
x
0
1
2
3
4



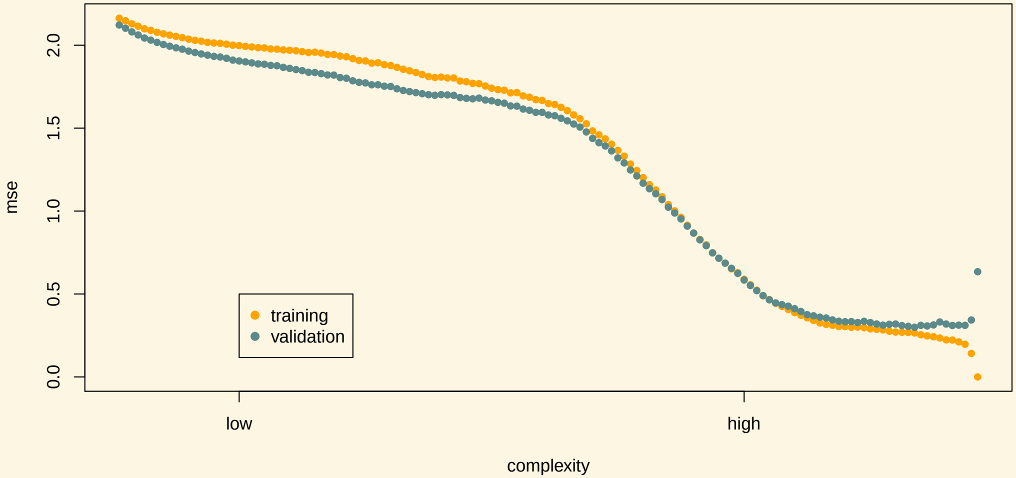
y
0
1
2
3
4
5
6
x
0
1
2
3
4

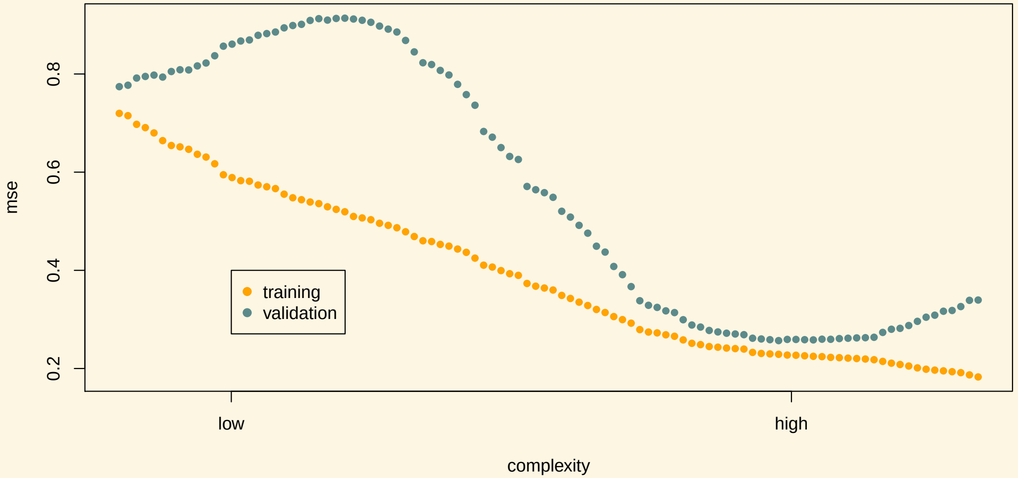
Tuning parameters, cont.

With this setup, we can now define the mean squared (validation) error:

$$MSE(k) = \frac{1}{\#\{V\}} \sum_{i \in V} \left(y_i - \widehat{g}_k(x_i)\right)^2$$

We can then optimize this function over k (or whatever tuning parameter is appropriate for the model of choice) to pick the final model.


mse
2.0
1.5
1.0
0.5
0.0
● training
● validation
low
high
complexity


mse
0.2
0.4
0.6
0.8
training
validation
low
high
complexity

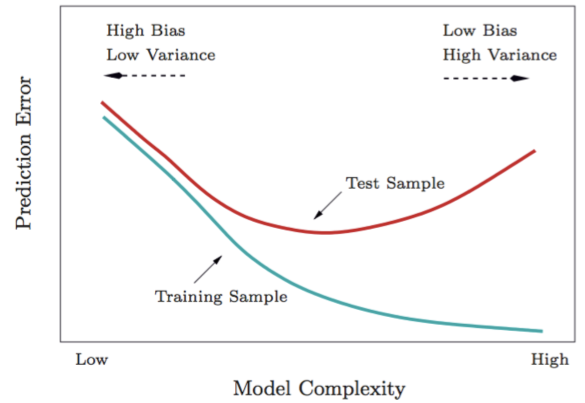


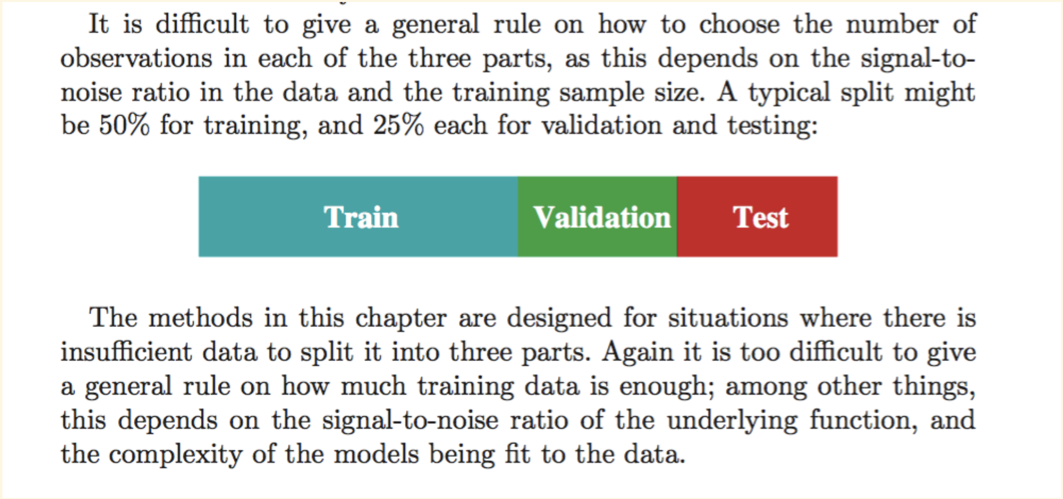
FIGURE 2.11. *Test and training error as a function of model complexity.*

Elements of Statistical Learning, pg. 38

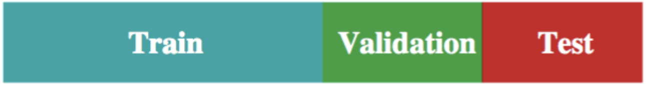
Evaluating the model

Now, it seems that it is sufficient to divide our dataset into two parts: testing and validation. However, we actually need a third chunk of data called the test set. Once the tuning parameters are selected based on the validation, we apply the model to the test set to judge how well our algorithm has performed.

Why do we need to do this instead of just using the MSE on the validation set?



It is difficult to give a general rule on how to choose the number of observations in each of the three parts, as this depends on the signal-to-noise ratio in the data and the training sample size. A typical split might be 50% for training, and 25% each for validation and testing:



The methods in this chapter are designed for situations where there is insufficient data to split it into three parts. Again it is too difficult to give a general rule on how much training data is enough; among other things, this depends on the signal-to-noise ratio of the underlying function, and the complexity of the models being fit to the data.

Elements of Statistical Learning, pg. 273

k-fold cross validation

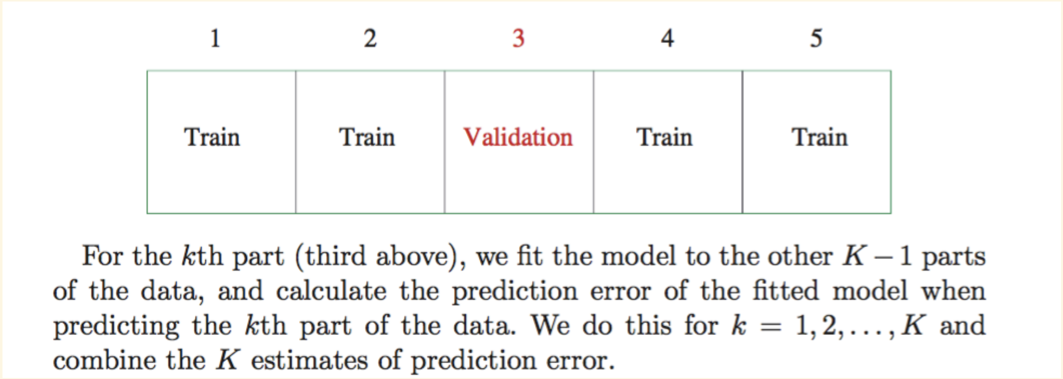
We always need a fixed test set, but there is an alternative to splitting the remainder of the data into distinct train and validation buckets.

k-fold cross validation

We always need a fixed test set, but there is an alternative to splitting the remainder of the data into distinct train and validation buckets.

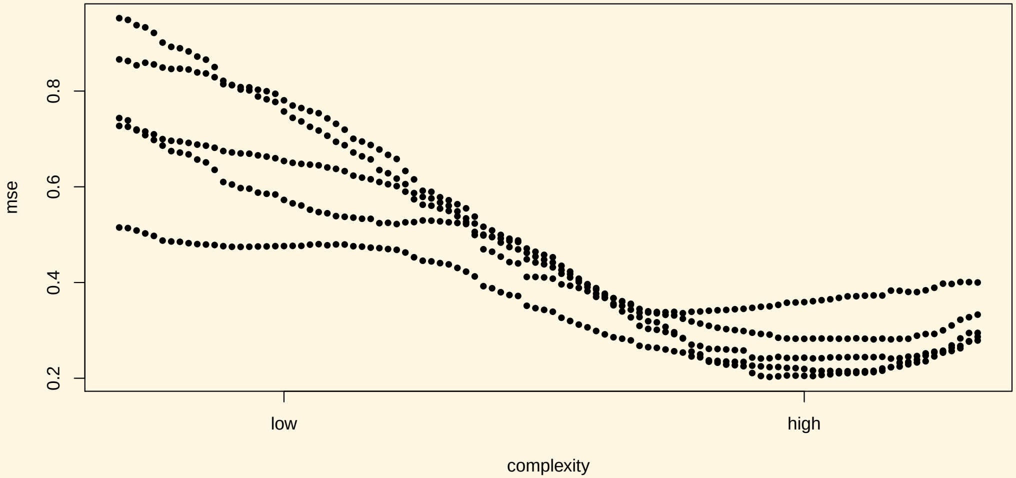
Cross validation, or more specifically k-fold cross validation instead:

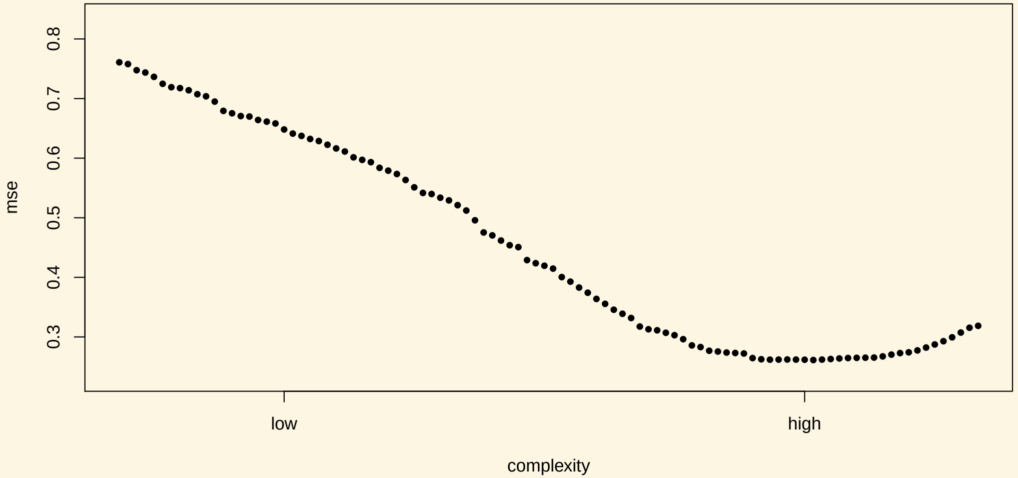
1. randomly partitions the training set into k equally sized buckets
2. select one bucket to be the validation set and calculates the MSE using the other $k - 1$ buckets to train the data
3. then, selects a different bucket to the validation set and trains on the remaining $k - 1$ buckets
4. this is repeated k times, until every observation has been in the validation set once
5. the MSE for any value of the tuning parameter is calculated by averaging the estimates across the k runs

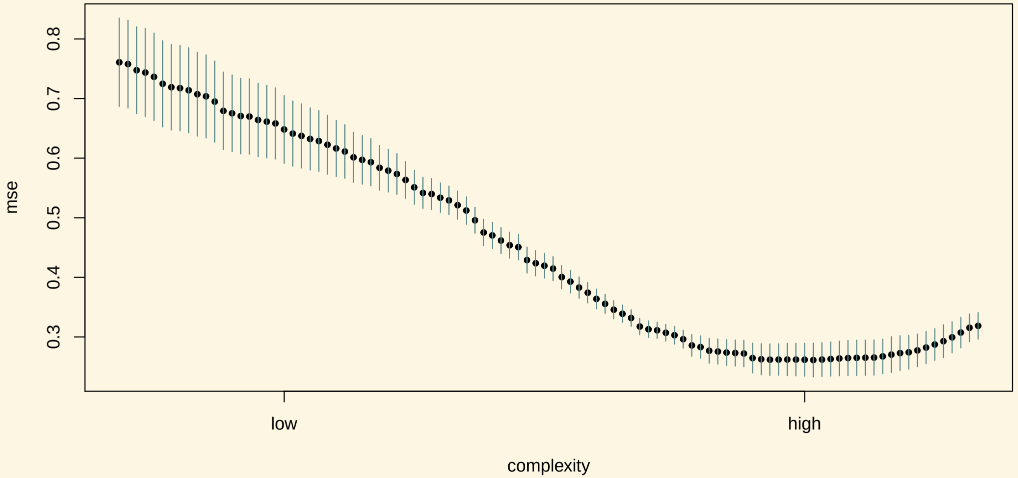

1
2
3
4
5
Train
Train
Validation
Train
Train

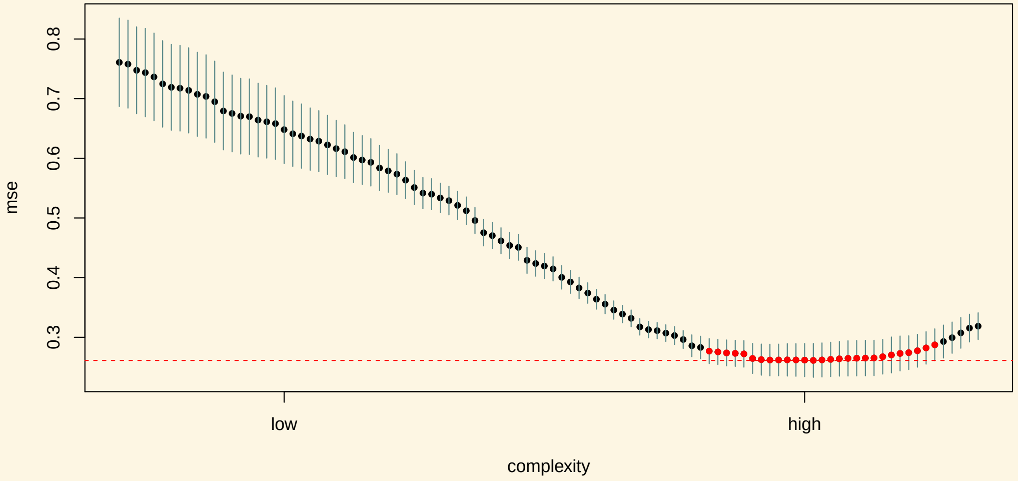
For the kth part (third above), we fit the model to the other $K-1$ parts of the data, and calculate the prediction error of the fitted model when predicting the kth part of the data. We do this for $k = 1, 2, \ldots, K$ and combine the K estimates of prediction error.

Elements of Statistical Learning, pg. 242


mse
0.2
0.4
0.6
0.8
low
high
complexity


mse
0.3
0.4
0.5
0.6
0.7
0.8
low
high
complexity


mse
0.3
0.4
0.5
0.6
0.7
0.8
low
high
complexity


mse
0.3
0.4
0.5
0.6
0.7
0.8
low
high
complexity

Still to come...

- what happens when we have higher dimensional spaces
- classification methods