Nonparametric Regression and Cross-Validation

Yen-Chi Chen

5/27/2017

Nonparametric Regression

In the regression analysis, we often observe a data consists of a response variable Y and a covariate X (this is the simplest case–we may have multiple covariate). The goal is to understand how the response Y and the covariate X is associated. Assume the sample size is n. We often use the notations

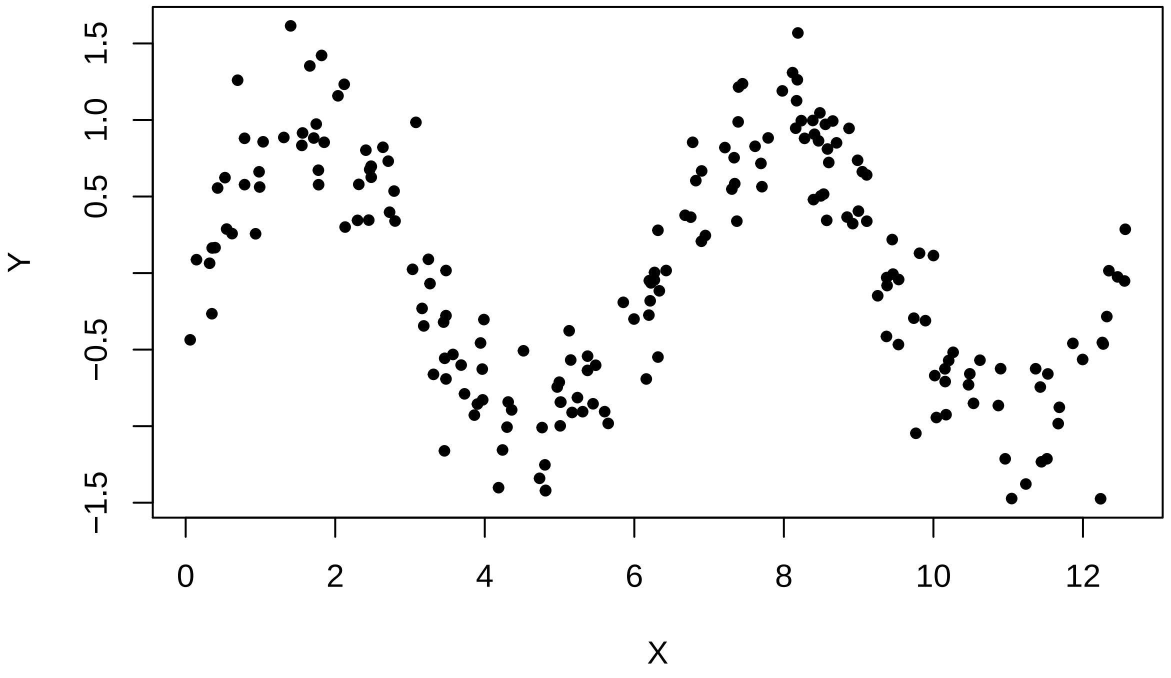
$$(X_1, Y_1), \cdots, (X_n, Y_n)$$

to denote the observe data and emphasizes the fact that one observation consists of a pair of a response Y_i and a covariate X_i. Formally, the regression analysis models the relationship between each pair (X_i, Y_i) as

$$Y_i = m(X_i) + \epsilon_i,$$

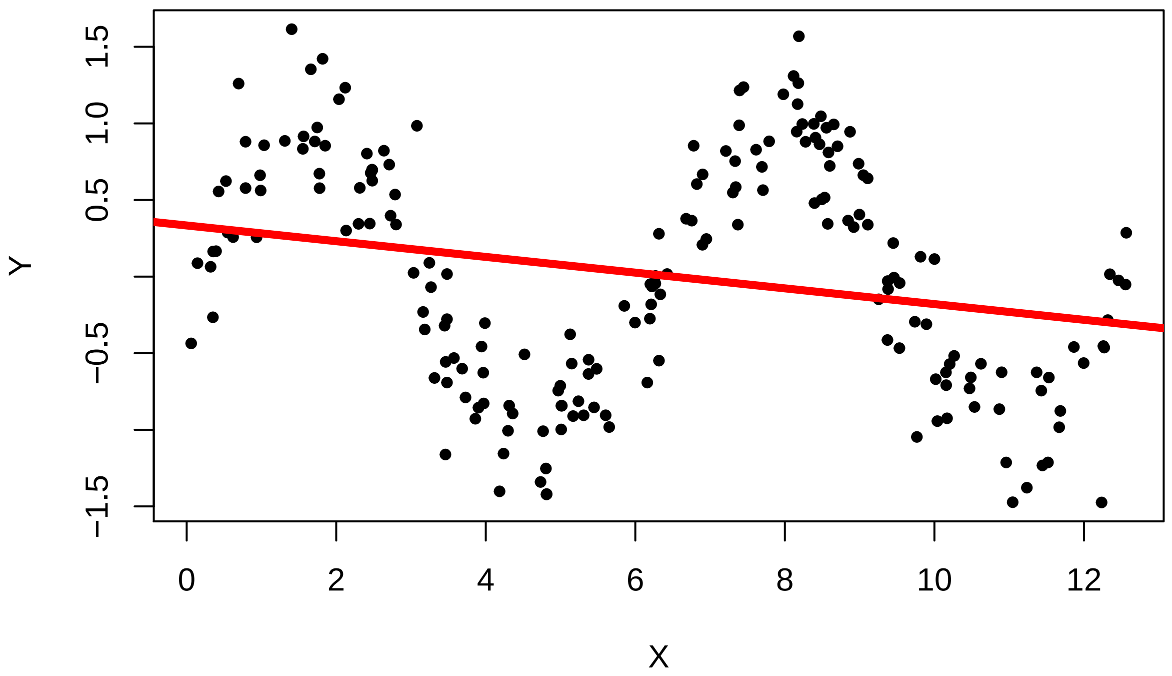
where ϵ_i is often modeled as a mean 0 Gaussian noise. The function $m(x)$ is called *regression function*. In the linear regression, we further assume that the regression function $m(x)$ is a linear function: $m(x) = \alpha + \beta x$, where α is the intercept and β is the slope. However, the actual relationship between X_i and Y_i may not be such a simple linear relation. Consider, for example, the following case:

```r
X = runif(200, min=0, max=4*pi)
Y = sin(X) + rnorm(200, sd=0.3)
plot(X,Y, pch=20)
```



The scatter plot shows a clear pattern between X and Y. What would happen if we apply the ordinary linear regression?

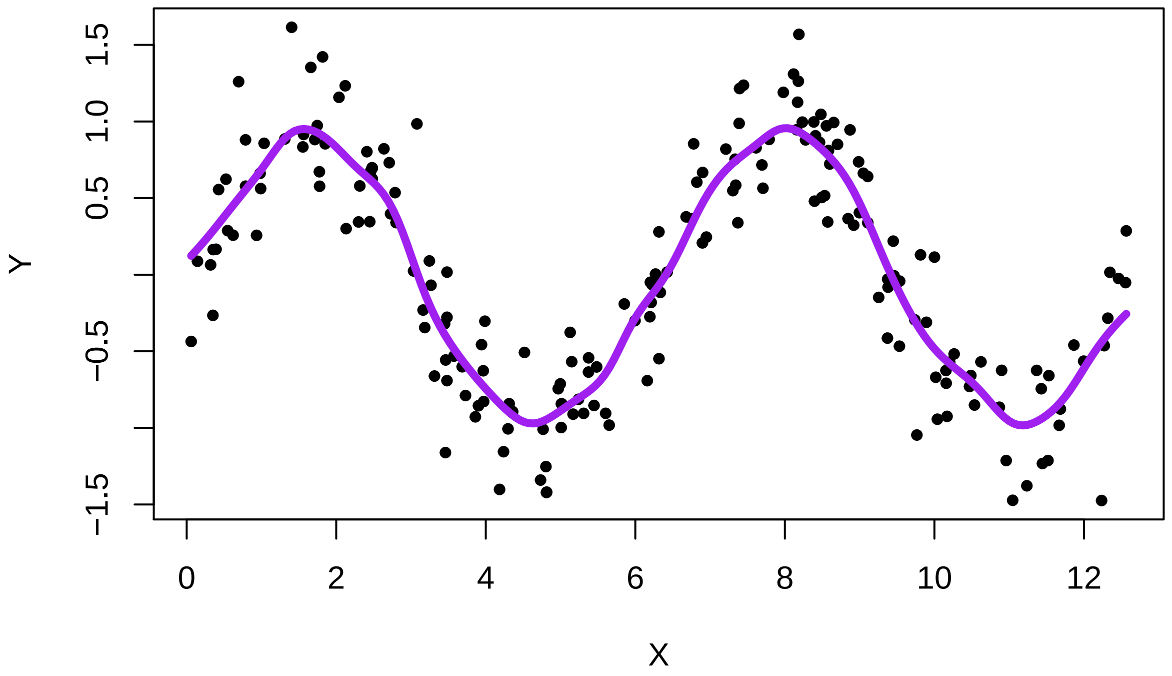
```r
fit = lm(Y~X)
plot(X,Y, pch=20)
abline(fit, lwd=4, col="red")
```



Well the linear regression (red line) does not capture the underlying structure. This is due to the fact that linear regression is only capable of detecting the linear relationship between the two variables. In this dataset, though we can observe a clear relation between X and Y but they have a non-linear relation, resulting in a bad outcome from the linear regression.

What can we do in this case? Well here is an approach called *kernel regression*, which is a *nonparametric regression method* that allows us to capture the underlying structure:

```
Kreg = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.9)
plot(X,Y,pch=20)
lines(Kreg, lwd=4, col="purple")
```

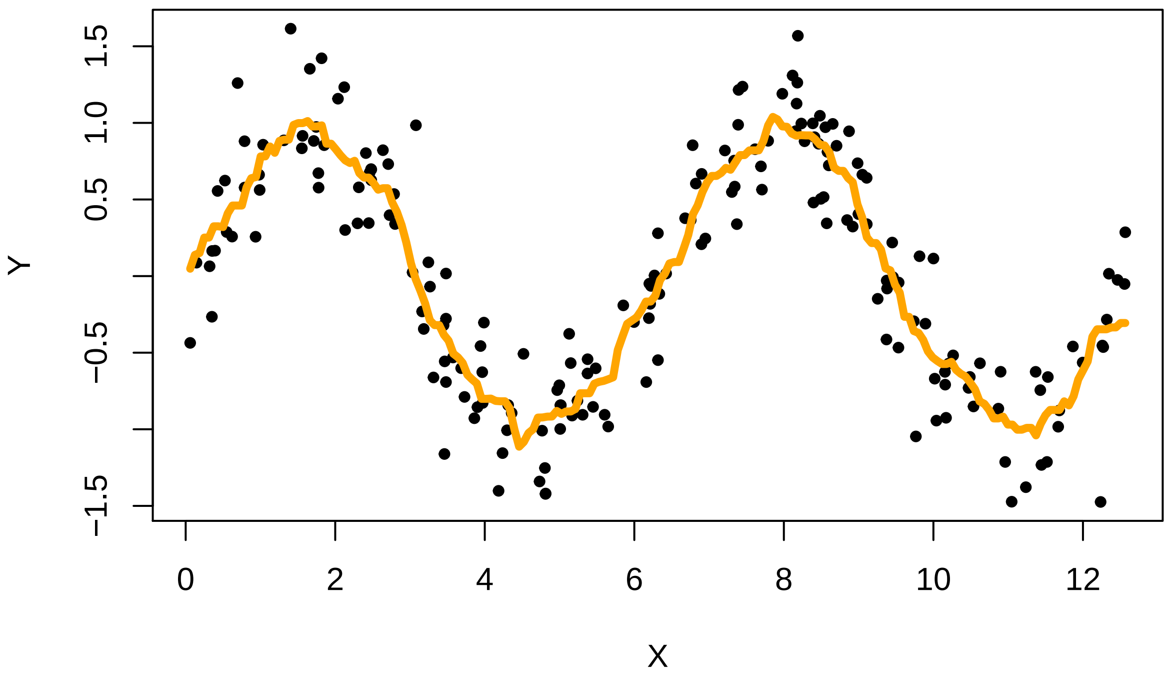


The orchid curve is the fitted value from the kernel regression. The idea of a kernel regression is that for a given point x, we then use a *weighted average* of the nearby points' response (Y) as the estimated value of the regression function. Points whose value of covariate (X) is closer to x will be given a higher weight. Observations whose covariate value are away from x will be given a lower weight (or even no weight). For example, at point $x = 4$, points whose covariate X is closer to 4 will be given a higher weight. We then estimate the regression function using the weight average of all the response.

There are two important configurations of the kernel regression: the kernel function (often be written as $K(x)$) and the smoothing bandwidth (often be written as h). These two quantities are similar to that of the kernel density estimator! The kernel function determines the decreasing pattern of the weight for observations whose covariate value is away from x. The smoothing bandwidth controls the decreasing rate. In a sense, we are using the kernel function and smoothing bandwidth to smooth out the data points to obtain a local estimate of the regression function.

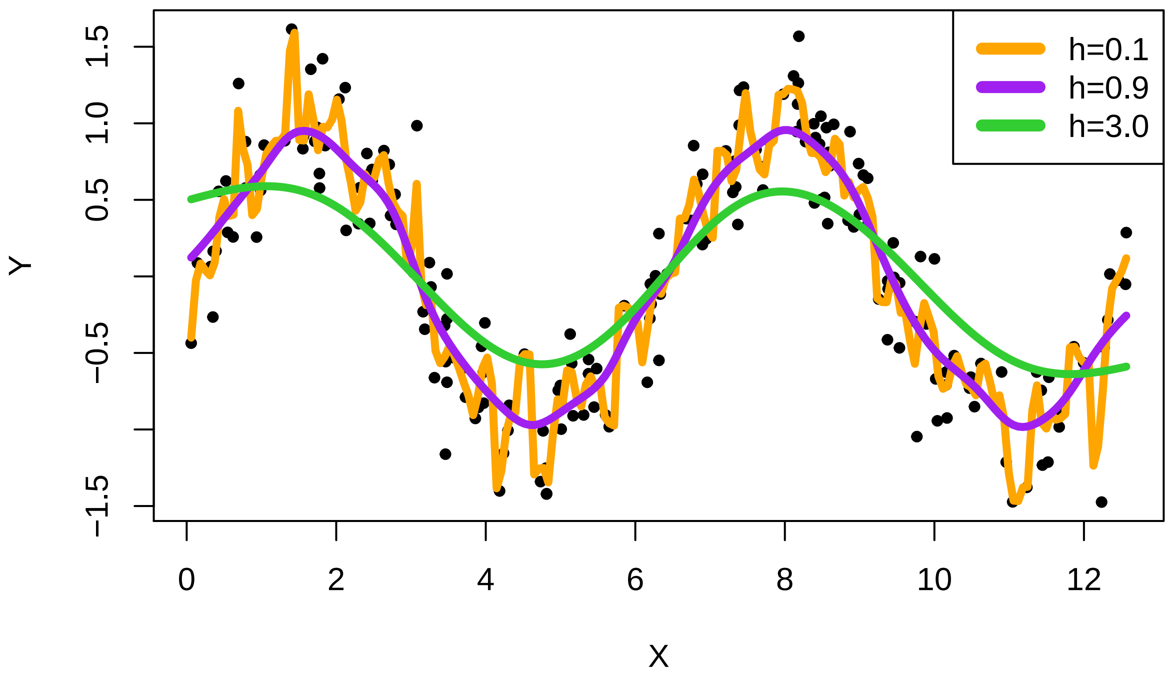
When we set `kernel="normal"`, we are using the Gaussian function as the kernel. An alternative is to choose `kernel="box"`; in this case, we will give equal weight to points whose distance to x is less than h:

```
Kreg = ksmooth(x=X,y=Y,kernel = "box",bandwidth = 0.9)
plot(X,Y,pch=20)
lines(Kreg, lwd=4, col="orange")
```



The argument `bandwidth` controls the amount of smoothing, just like the on in the kernel density estimator. Here are examples of changing the value of `bandwidth`:

```
Kreg1 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.1)
Kreg2 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 0.9)
Kreg3 = ksmooth(x=X,y=Y,kernel = "normal",bandwidth = 3.0)
plot(X,Y,pch=20)
lines(Kreg1, lwd=4, col="orange")
lines(Kreg2, lwd=4, col="purple")
lines(Kreg3, lwd=4, col="limegreen")
legend("topright", c("h=0.1","h=0.9","h=3.0"), lwd=6, col=c("orange","purple","limegreen"))
```



Thus, you may guess that this kernel regression also suffers from the *bias-variance* tradeoff: when h is small (the orange curve), the variabiility is large but the bias is small; when h is large (green curve), the variability is small but the bias is large!

Here is a code that you can run to see how the variability and bias are interacting with each other:

```
h0 = 0.2
  # try to change h0 to other values and see what happen
for(i in 1:100){
  X1 = runif(200, min=0, max=4*pi)
  Y1 = sin(X1) + rnorm(200, sd=0.3)
  Kreg = ksmooth(x=X1,y=Y1,kernel = "normal",bandwidth = h0)
  plot(X1,Y1,pch=20, main=paste("h =",h0))
  lines(Kreg, lwd=4, col="purple")
  Sys.sleep(0.5)
}
```

The best smoothing bandwidth should balance both the bias and variability. In the next section, we will introduce an elegant approach called *cross-validation* that allows us to choose the smoothing bandwidth subject to certain optimality criterion.

Exercise 1:

1. Simulate 500 pairs of (X, Y) such that

$$Y_i = e^{-0.1\cdot X_i}\cos(X_i) + \epsilon_i, \quad X_i \sim \mathsf{Unif}[0, 6\pi], \quad \epsilon \sim N(0, 0.1^2).$$

   Namely, we first generate X_i from an uniform distribution over $[0, 6\pi]$ and then generate Y_i by adding up $e^{-0.1X_i}\cos(X_i)$ and a random noise from a normal distribution with mean 0 and SD 0.1.
2. Show a scatter plot of X, Y. Fit a kernel regression with smoothing bandwidth $h = 0.5$ and Gaussian kernel (`kernel = 'normal'`). Attach the fitted regression curve to the scatter plot.
3. Consider three smoothing bandwidth $h = 0.1, 0.5, 2.0$. Fit the kernel regression to each of them and attach the three curves to the scatter plot. Which smoothing bandwidth seems to be the best one?

4. Now return to the first question, change the SD of the noise from 0.1 to 0.5 (this SD is sometimes called the noise level) and redo question 3. Which smoothing bandwidth seems to be the best one?

Cross-Validation

The cross-validation (CV) is a method to *estimate* the predicion error of our regression estimator. The idea is very simple: we split the data into two parts, one part is called the *training set* and the other is called the *validation set*. We then use the data in the training set to construct our estimator and predict the value of response on the validation set and then calculate the quality of our prediction. The reason of splitting the data into two parts is to *avoid the problem of using the data twice* (this is related to the problem of *overfitting*).

A simplest form of cross-validation is the *leave-one-out cross-validation* (LOOCV). The idea is: each time we leave one observation out as the validation set and we use the remaining data points to fit the data and then predict the value that we left out. Here is what we write in R:

```
n = length(X)
  # n: sample size
CV_err = rep(NA, n)
for(i in 1:n){
  X_val = X[i]
  Y_val = Y[i]
    # validation set
  X_tr = X[-i]
  Y_tr = Y[-i]
    # training set
  Y_val_predict = ksmooth(x=X_tr,y=Y_tr,kernel = "normal",bandwidth=0.9,
                          x.points = X_val)
  CV_err[i] = (Y_val - Y_val_predict$y)^2
    # we measure the error in terms of difference sqaure
}
mean(CV_err)
```

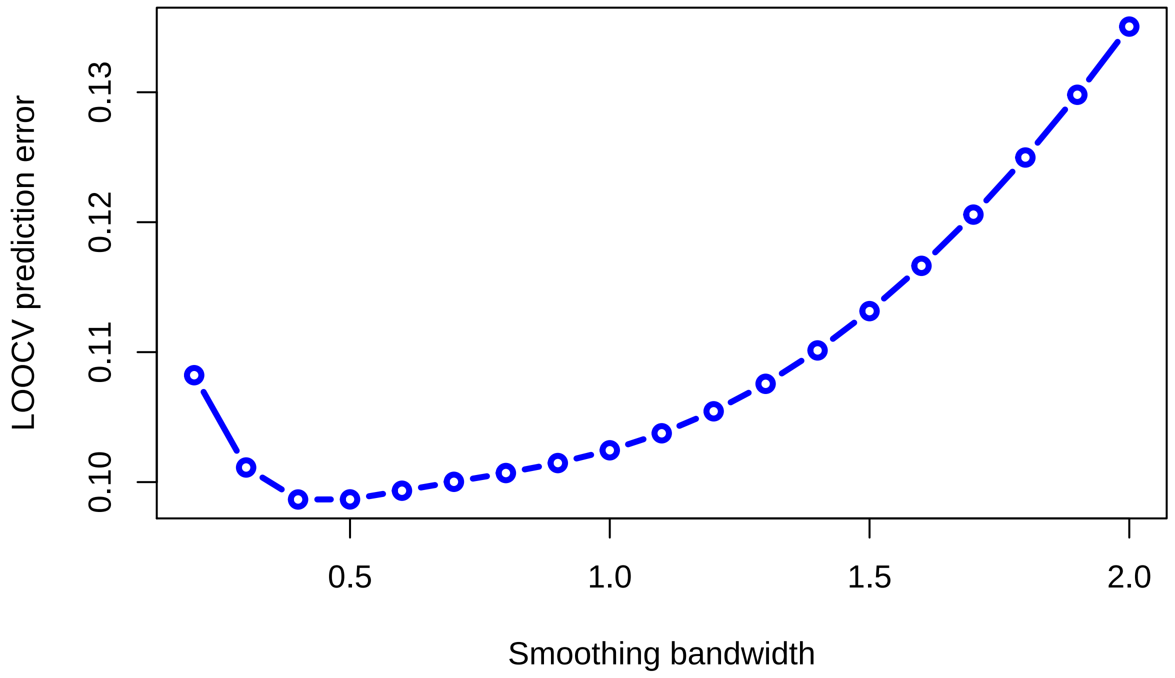
```
## [1] 0.1014641
```

The output `mean(CV_err)` gives an estimate of the prediction error using the smoothing bandwidth $h = 0.9$. Now to see how the smoothing bandwidth h changes the quality of prediction, we will apply this procedure to many possible values of h.

```
n = length(X)
  # n: sample size
h_seq = seq(from=0.2,to=2.0, by=0.1)
  # smoothing bandwidths we are using
CV_err_h = rep(NA,length(h_seq))
for(j in 1:length(h_seq)){
  h_using = h_seq[j]
  CV_err = rep(NA, n)
  for(i in 1:n){
    X_val = X[i]
    Y_val = Y[i]
      # validation set
    X_tr = X[-i]
    Y_tr = Y[-i]
      # training set
    Y_val_predict = ksmooth(x=X_tr,y=Y_tr,kernel = "normal",bandwidth=h_using,
                                x.points = X_val)
    CV_err[i] = (Y_val - Y_val_predict$y)^2
      # we measure the error in terms of difference square
  }
  CV_err_h[j] = mean(CV_err)
}
CV_err_h
```

```
##  [1] 0.10822694 0.10112420 0.09866272 0.09867038 0.09933617 0.10002225
##  [7] 0.10069739 0.10146408 0.10244873 0.10375680 0.10544723 0.10756305
## [13] 0.11013240 0.11315749 0.11664110 0.12058279 0.12497659 0.12980483
## [19] 0.13505029
```

```
plot(x=h_seq, y=CV_err_h, type="b", lwd=3, col="blue",
     xlab="Smoothing bandwidth", ylab="LOOCV prediction error")
```



Here we see a beautiful curve, indicating how the smoothing bandwidth affects the quality of prediction. We can then choose the smoothing bandwidth that minimizes this LOOCV error:

```
which(CV_err_h == min(CV_err_h))
```

```
## [1] 3
```

```
  # this gives us the index of the minimum error

h_seq[which(CV_err_h == min(CV_err_h))]
```

```
## [1] 0.4
```

```
  # this is the smoothing bandwidth leading to the minimum LOOCV error
```

There are many variants of the cross-validatioin. For instance, generalized cross-validation, k-fold cross-validation (k-CV). In particular, k-CV seems to be the most common one used in practice. The steps of k-CV is as follows.

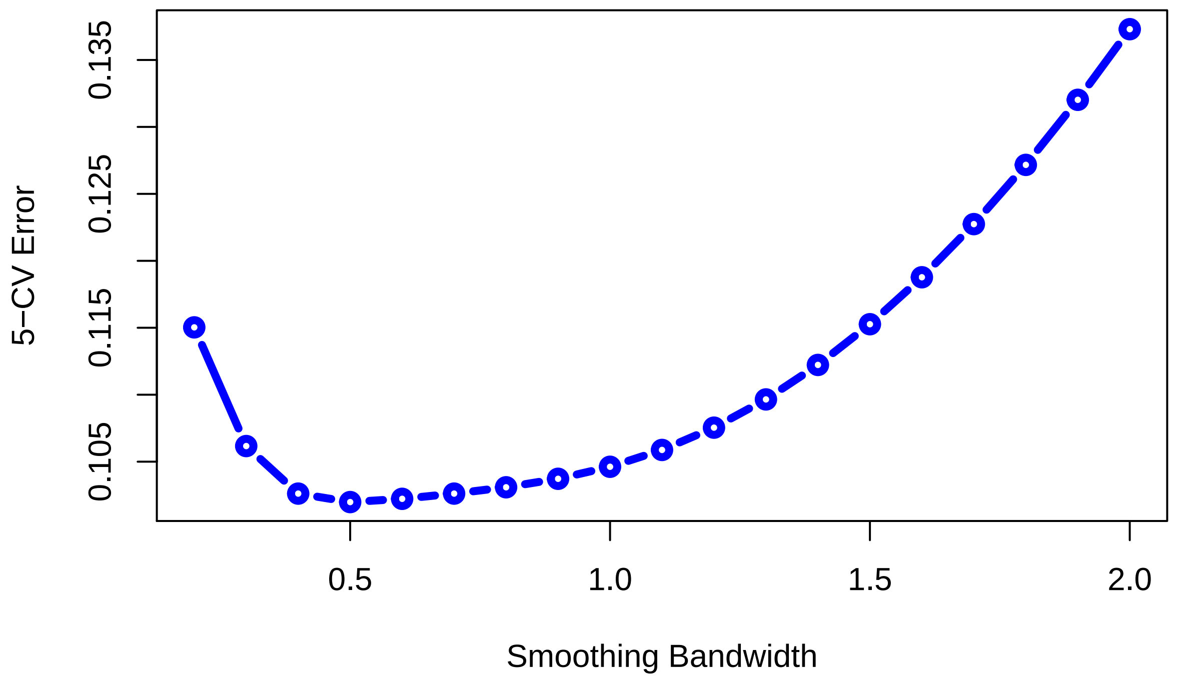
1. We random split the data into k roughly equal size subsamples, called them $D_1, \cdots, D_k$.

2. We use D_1 as the validation set and others as training set, compute the predition error, called it Err_1.
3. Then we use D_2 as the validation set and others as training set, compute the predition error Err_2.
4. We keep doing this until every subsample has been used as a validation set.
5. We use the average of these k prediction errors $Err = \frac{1}{k}\sum_{\ell=1}^{k} Err_\ell$ as an estimate of the prediction error.
6. After obtaining one such an estimate, we will repeat step 1-5 several times (re-split the data into k subsamples, computing the prediction error) and use the average prediction errors as the final error estimate.

Here is an example of 5-CV:

```
n = length(X)
N_cv = 100
k = 5
cv_lab = sample(n,n,replace=F) %% k
  ## randomly split all the indices into k numbers
h_seq = seq(from=0.2,to=2.0, by=0.1)

CV_err_h = rep(0,length(h_seq))
for(i_tmp in 1:N_cv){
  CV_err_h_tmp = rep(0, length(h_seq))
  cv_lab = sample(n,n,replace=F) %% k
  for(i in 1:length(h_seq)){
    h0 = h_seq[i]
    CV_err =0
    for(i_cv in 1:k){
      w_val = which(cv_lab==(i_cv-1))
      X_tr = X[-w_val]
      Y_tr = Y[-w_val]
      X_val = X[w_val]
      Y_val = Y[w_val]
      kernel_reg = ksmooth(x = X_tr,y=Y_tr,kernel = "normal",bandwidth=h0,
                           x.points=X_val)
        # WARNING! The ksmooth() function will order the x.points from
        # the smallest to the largest!
        CV_err = CV_err+mean((Y_val[order(X_val)]-kernel_reg$y)^2,na.rm=T)
        # na.rm = T: remove the case of 'NA'
      }
    CV_err_h_tmp[i] = CV_err/k
  }
  CV_err_h = CV_err_h+CV_err_h_tmp
}
CV_err_h = CV_err_h/N_cv
plot(h_seq,CV_err_h, type="b", lwd=4, col="blue", xlab="Smoothing Bandwidth",
     ylab="5-CV Error")
```



```
h_opt = h_seq[which(CV_err_h==min(CV_err_h))]
h_opt
```

```
## [1] 0.5
```

Compare to the LOOCV, you would find that both approachs show a quiet similar pattern. By the way, you may think about why the optimal value of CV error is about 0.1. This is because when we design the simulation, our model is

$$Y_i = \sin(X_i) + \epsilon_i, \quad \epsilon_i \sim N(0, 0.3^2).$$

Thus, the variance of the noise is $0.3^2 = 0.09$. This value 0.09 will be the *best prediction error* that any estimator can achieve in the long run (this is the error corresponds to the estimator being the same as the true generating function).

Exercise 2:

1. Simulate again 500 pairs of (X, Y) such that

$$Y_i = e^{-0.1 \cdot X_i} \cos(X_i) + \epsilon_i, \quad X_i \sim \mathsf{Unif}[0, 6\pi], \quad \epsilon \sim N(0, 0.1^2).$$

2. Use the leave-one out cross-validation to show the smoothing bandwidth versus CV-error plot for the kernel regression. Which smoothing bandwidth provides you the minimal CV-error? (I would recommend searching the range of h over $[0.1, 2.0]$)
3. Show the scatter plot along with the kernel regression using the optimal smoothing bandwidth.
4. Change the SD in the noise from 0.1 to 0.5 and redo question 1 and 2. Does the optimal smoothing bandwidth change?

Smoothing Spline

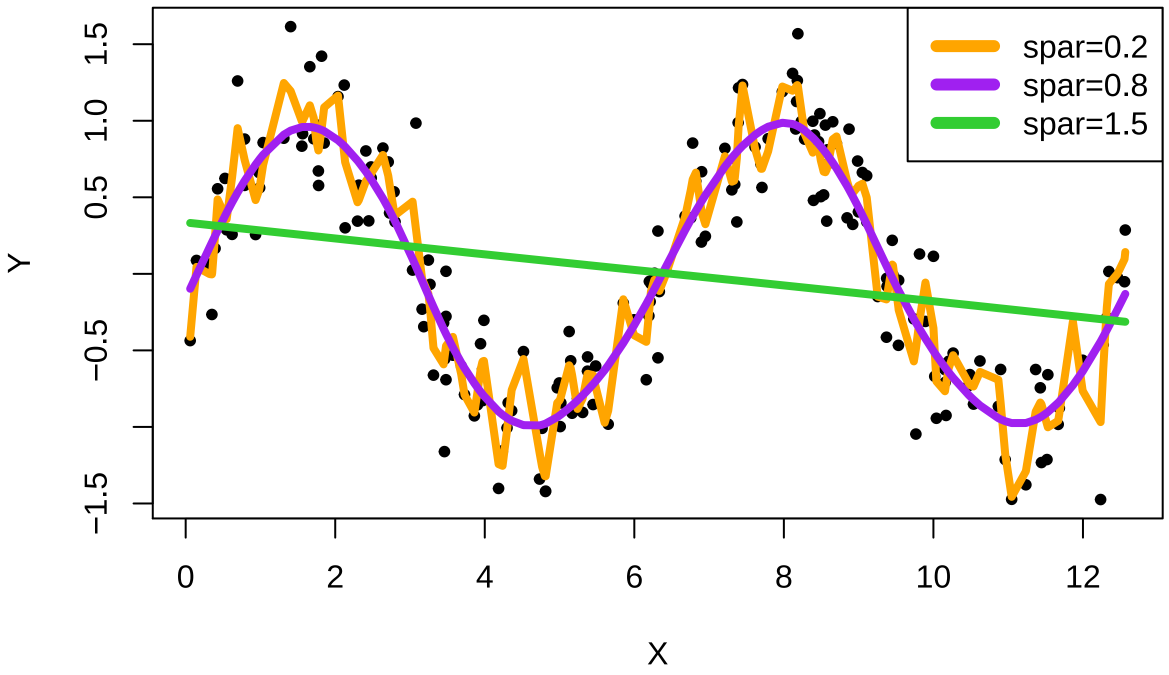
Here we introduce another nonparametric method: the *smoothing spline* approach. The idea of smoothing spline is: we want to find a function $f(x)$ that fits the data well but also is very smooth. Using a more mathematical definition, we want to find a function $f(x)$ such that

$$\sum_{i=1}^{n}(Y_i - f(X_i))^2 + \lambda \int_{X_{\min}}^{X_{\max}} |f''(x)|^2 dx$$

is minimized. The quantity λ is quantity that is called smoothness penalty (or smoothing parameter in some literature) that determines how much we will loss if we allow the function to adapt to the data. Without this part of penalty, the best function f will be the ones passing through every observation. Note that the $X_{\min}$ and $X_{\max}$ are the minimum and maximum value of all the observed covariates.

Here are some examples of fitting the smoothing spline to the data under different values of λ. In R, we use the function `smooth.spline()` to fit a smoothing spline and the argument `spar` is how we can specify the value of λ in the above criterion (note that here is a transformation from `spar` to the λ; see the manual of this function):

```
SS1 = smooth.spline(x=X,y=Y,spar=0.2)
SS2 = smooth.spline(x=X,y=Y,spar=0.8)
SS3 = smooth.spline(x=X,y=Y,spar=1.5)
plot(X,Y,pch=20)
lines(SS1, lwd=4, col="orange")
lines(SS2, lwd=4, col="purple")
lines(SS3, lwd=4, col="limegreen")
legend("topright", c("spar=0.2","spar=0.8","spar=1.5"), lwd=6,
       col=c("orange","purple","limegreen"))
```



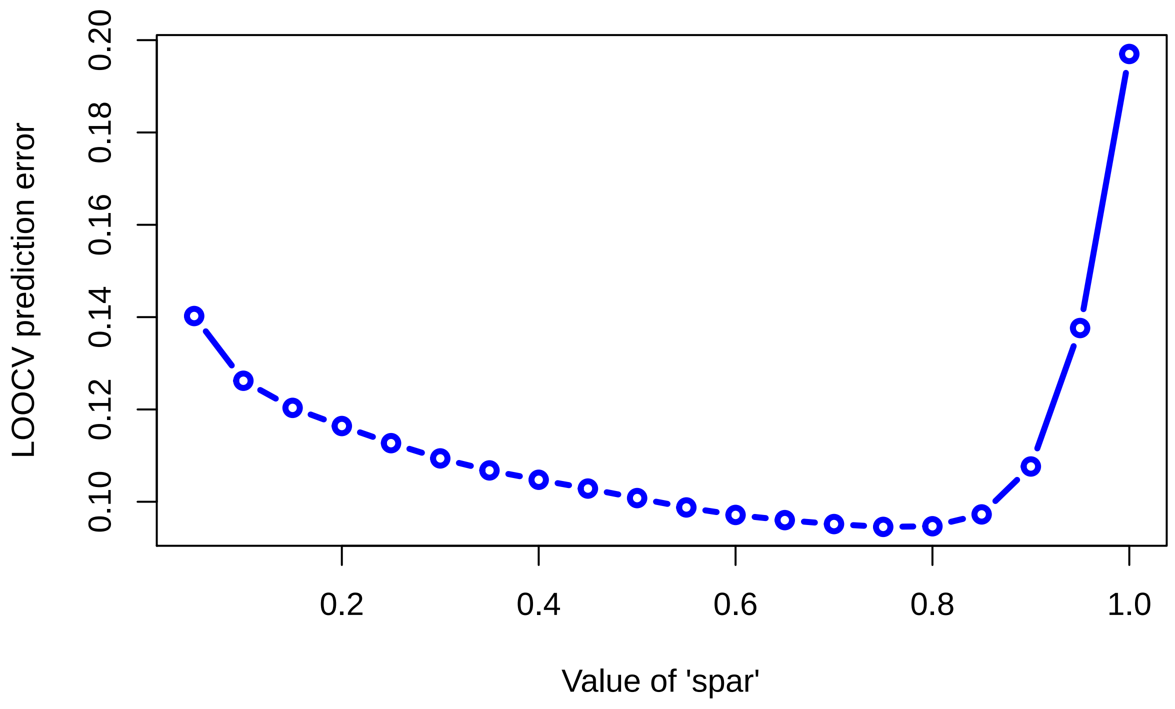
Similar to the kernel regression, we can also use the idea of cross-validation to determine which value of `spar` we should use. Here is how we implement it in practice:

```
n = length(X)
  # n: sample size
sp_seq = seq(from=0.05,to=1.0, by=0.05)
  # values of spar we are exploring
CV_err_sp = rep(NA,length(sp_seq))
for(j in 1:length(sp_seq)){
  spar_using = sp_seq[j]
```

```r
  CV_err = rep(NA, n)
  for(i in 1:n){
    X_val = X[i]
    Y_val = Y[i]
      # validation set
    X_tr = X[-i]
    Y_tr = Y[-i]
      # training set
    SS_fit = smooth.spline(x=X_tr,y=Y_tr,spar=spar_using)
    Y_val_predict = predict(SS_fit,x=X_val)
      # we use the 'predict()' function to predict a new value
    CV_err[i] = (Y_val - Y_val_predict$y)^2
  }
  CV_err_sp[j] = mean(CV_err)
}
CV_err_sp
```

```
##  [1] 0.14023770 0.12621731 0.12034069 0.11640628 0.11269646 0.10939354
##  [7] 0.10679102 0.10476483 0.10285855 0.10079543 0.09876707 0.09714631
## [13] 0.09602409 0.09517049 0.09455990 0.09469657 0.09726382 0.10764380
## [19] 0.13763385 0.19700140
```

```r
plot(x=sp_seq, y=CV_err_sp, type="b", lwd=3, col="blue",
     xlab="Value of 'spar'", ylab="LOOCV prediction error")
```



```r
sp_seq[which(CV_err_sp == min(CV_err_sp))]
```

```
## [1] 0.75
```