SDS 383D, Exercises 3: Linear smoothing and Gaussian processes

Jan-Michael Cabrera

March 11, 2019

Curve fitting by linear smoothing

(A) Linear smoothing Suppose we want to estimate the value of the regression function y^* at some new point x^*, denoted $\hat{f}(x^*)$. Assume for the moment that $f(x)$ is linear, and that y and x have already had their means subtracted, in which case $y_i = \beta x_i + \epsilon_i$.

Show that for the one-predictor case, your prediction $\hat{y}^* = \hat{f}(x^*) = \hat{\beta}x^*$ may be expressed as:

$$\hat{f}(x^*) = \sum_{i=1}^{n} w(x_i, x^*) y_i$$

We can express $\hat{\beta}$ in matrix notation as,

$$\hat{\beta} = (X^T X)^{-1} X^T Y.$$

Plugging this in to find our predictor, $\hat{y}^*$ we get,

$$\begin{aligned} \hat{y}^* &= x^* (X^T X)^{-1} X^T Y \\ &= \frac{X^T Y}{X^T X} \\ &= \frac{\sum_{i=1}^{n} x_i x^* y_i}{\sum_{i=1}^{n} x_i^2} \end{aligned}$$

The result is of the form $\hat{f}(x^*) = \sum_{i=1}^{n} w(x_i, x^*) y_i$ with the weights expressed as,

$$w(x_i, x^*) = \frac{x_i x^*}{\sum_{i=1}^{n} x_i^2}.$$

This resultant *smoother* has constant weights and produces predictions that lie on the line with slope $\hat{\beta}$. If the data is binned and locality preserved, this estimate is similar to a running line average.

$$w_K(x_i, x^*) = \begin{cases} 1/K, & x_i \text{ one of the } K \text{ closest sample points to } x^*, \\ 0, & \text{otherwise.} \end{cases}$$

With *K-nearest-neighbor smoothing* $\hat{y}^*$ is essentially the arithmetic mean of the K y_i's nearest x^*.

(B) A *kernel function* $K(x)$ is a smooth function satisyfing

$$\int_{\mathbb{R}} K(x)dx = 1\,, \quad \int_{\mathbb{R}} xK(x)dx = 0\,, \quad \int_{\mathbb{R}} x^2 K(x)dx > 0\,.$$

A very simple example is the uniform kernel,

$$K(x) = \frac{1}{2} I(x) \quad \text{where} \quad I(x) = \begin{cases} 1, & |x| \leq 1 \\ 0, & \text{otherwise}. \end{cases}$$

Another common example is the Gaussian kernel:

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Kernels are used as weighting functions for taking local averages. Specifically, define the weighting function

$$w(x_i, x^\star) = \frac{1}{h} K\left(\frac{x_j - x^\star}{h}\right),$$

where h is the bandwidth. Using this weighting function in a linear smoother is called *kernel regression*. (The weighting function gives the unnormalized weights; you should normalize the weights so that they sum to 1.)

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.append('../../scripts/')
from smoothers import kernel_smoother
```

```
# Create noisy data, y_i = sin(x_i) + e_i
x = np.linspace(0, 2*np.pi, num = 20)
y = np.sin(x) + np.random.normal(scale=0.2, size=x.shape)

# Create vector for fitting purposes
x_star = np.linspace(0, 2*np.pi)

# Instantiate array of bandwidths
h = np.array([0.25, 1.0])
```

```
# Instantiate a list to append kernel_smoother objects
H = []

# Iterates through array of bandwidths and passes the feature vector, response vector, and ↩
    bandwidth to kernel_smoother object
for i in range(len(h)):
    H.append(kernel_smoother(x, y, x_star, h=h[i]))
```

```
# Plots data
plt.figure()
plt.plot(x, y, '.k', label='Noisy response')
plt.plot(x, np.sin(x), '--k', label='True function')
# Iterates over the smoother objects and plots functions for the uniform and gaussian kernels
for i in range(len(h)):
    plt.plot(x_star, H[i].predictor(kernel='uniform'), label='Uniform Kernel, h='+str(h[i]))
    plt.plot(x_star, H[i].predictor(kernel='gaussian'), label='Gaussian Kernel, h='+str(h[i]))
plt.legend(loc=0)
# plt.show()
plt.savefig('figures/kernel_smoother.pdf')
plt.close()
```

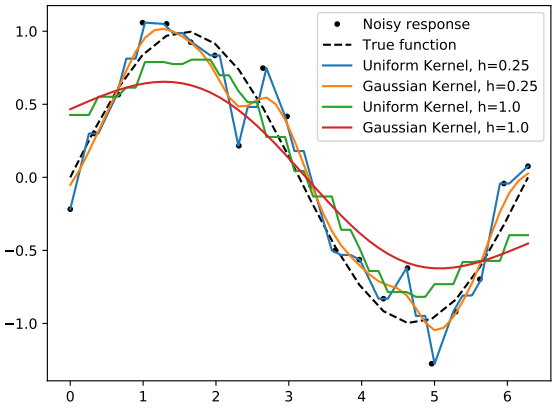

Figure 1: Uniform and Gaussian Kernel smoothing for different bandwidths

Cross validation

(A) Presumably a good choice of h would be one that led to smaller predictive errors on fresh data. Write a function or script that will: (1) accept an old ("training") data set and a new ("testing") data set as inputs; (2) fit the kernel-regression estimator to the training data for specified choices of h; and (3) return the estimated functions and the realized prediction error on the testing data for each value of h. This should involve a fairly straightforward "wrapper" of the function you've already written.

```
from __future__ import division
import numpy as np
import sys
sys.path.append('../../scripts/')
from smoothers import kernel_smoother
```

```
# Function for testing smoothers
def func(x):
    return np.sin(x)

# Initialize random seed
np.random.seed(3)

# Initialize x-vector
x = np.linspace(0, 2*np.pi, num=20)

# Training data of the form, y_i = f(x_i) + \epsilon_i
y_training = func(x) + np.random.normal(scale=0.2, size=x.shape)

# Testing data of the form, y_i = f(x_i) + \epsilon_i
y_test = func(x) + np.random.normal(scale=0.2, size=x.shape)
```

```
# Initialize list for storing kernel_smoother objects
H =[]

# Array of differing bandwidths to evaluate
h = np.array([0.2, 0.3, 0.4, 0.5, 0.6])

# Initialize MSE vector
mse = np.zeros(len(h))
```

```
# Iterates over bandwidth array and finds approximate MSE for each
for i in range(len(h)):
    H.append(kernel_smoother(x, y_training, x, h=h[i]))
    H[i].predictor()
    mse[i] = H[i].MSE(y_test)

print(mse)
```

```
jancabrera@phoenix$ python cross_validation_a.py
[0.07662882 0.0815833  0.0892882  0.10014996 0.11362704]
```

(B) Imagine a conceptual two-by-two table for the unknown, true state of affairs. The rows of the table are "wiggly function" and "smooth function," and the columns are "highly noisy observations" and "not so noisy observations." Simulate one data set (say, 500 points) for each of the four cells of this table, where the x's take values in the unit interval. Then split each data set into training and testing subsets. You

choose the functions. Apply your method to each case, using the testing data to select a bandwidth parameter. Choose the estimate that minimizes the average squared error in prediction, which estimates the mean-squared error:

$$L_n(\hat{f}) = \frac{1}{n}\sum_{i=1}^{n^\star}(y_i^\star - \hat{y}_i^\star)^2\,,$$

where $(y_i^\star, x_i^\star)$ are the points in the test set, and $\hat{y}_i^\star$ is your predicted value arising from the model you fit using only the training data. Does your out-of-sample predictive validation method lead to reasonable choices of h for each case?

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.append('../../scripts/')
from smoothers import kernel_smoother
```

```
# Function for testing smoothers
def func(x, period = 1):
    return np.sin(x*2*np.pi*period)

# Initialize random seed
np.random.seed(3)

# Initialize x-vector
x = np.linspace(0, 1, num=100)

# Noise level array, low and high
noise = np.array([0.05, 0.25])
noise_label = ['Low', 'High']

# Period of function, high period (wiggly), low period (smooth)
period = np.array([3, 0.5])
period_label = ['Wiggly', 'Smooth']

### Initialize lists for storing the different responses
# Training responses
Y_training = []
# Testing responses
Y_test = []
# True response
T = []
```

```
# Iterates over noise and period arrays to assign data to lists
for n in range(len(noise)):
    for p in range(len(period)):
        Y_training.append(func(x, period=period[p]) + np.random.normal(scale=noise[n], size=x.↩
            shape))
        Y_test.append(func(x, period=period[p]) + np.random.normal(scale=noise[n], size=x.↩
            shape))
        T.append(func(x, period=period[p]))
```

```
# Generates plots
i = 0
for n in range(len(noise)):
```

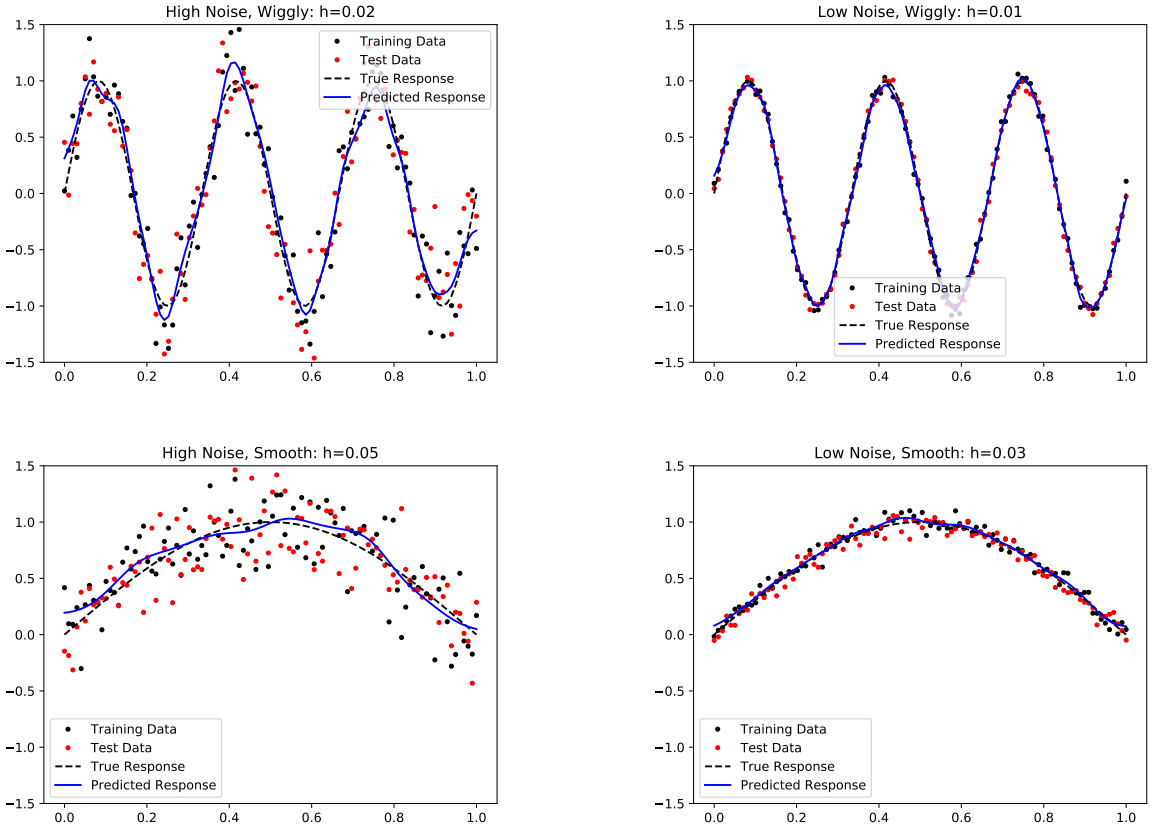

Figure 2: Bandwidths for various functions and noise levels

```
    for p in range(len(period)):

        # Instantiate kernel_smoother object
        y_smooth = kernel_smoother(x, Y_training[i], x)
        # Perform initial curve fit
        y_smooth.predictor()

        # Optimize bandwidth given the test data
        y_smooth.optimize_h(Y_test[i])

        # Perform curve fit with optimized bandwidth
        y_smooth.predictor()

        # Predicted values for opimized bandwidth
        y_pred = y_smooth.y_star
        h_star = y_smooth.h

        plt.figure()
        plt.title(noise_label[n]+' Noise, '+ period_label[p] + ': h={:.2f}'.format(h_star[0]))
        plt.plot(x, Y_training[i], '.k', label='Training Data')
        plt.plot(x, Y_test[i], '.r', label='Test Data')
        plt.plot(x, T[i], '--k', label='True Response')
        plt.plot(x, y_pred, '-b', label='Predicted Response')
        plt.legend(loc=0)
        # plt.show()
        plt.savefig('figures/cross_validation_'+noise_label[n]+'_'+period_label[p]+'.pdf')
        plt.close()
        i += 1
```

Local polynomial regression

Kernel regression has a nice interpretation as a "locally constant" estimator, obtained from locally weighted least squares. To see this, suppose we observe pairs (x_i, y_i) for $i = 1, \ldots, n$ from our new favorite model, $y_i = f(x_i) + \epsilon_i$ and wish to estimate the value of the underlying function $f(x)$ at some point x by weighted least squares. Our estimate is the scalar quantity

$$\hat{f}(x) = a = \arg\min_{\mathbb{R}} \sum_{i=1}^{n} w_i (y_i - a)^2 ,$$

where the w_i are the normalized weights (i.e. they have been rescaled to sum to 1 for fixed x). Clearly if $w_i = 1/n$, the estimate is simply $\bar{y}$, the sample mean, which is the "best" globally constant estimator. Using elementary calculus, it is easy to see that if the unnormalized weights are

$$w_i \equiv w(x, x_i) = \frac{1}{h} K\left(\frac{x_i - x}{h}\right) ,$$

then the solution is exactly the kernel-regression estimator.

(A) A natural generalization of locally constant regression is local polynomial regression. For points u in a neighborhood of the target point x, define the polynomial

$$g_x(u; a) = a_0 + \sum_{k=1}^{D} a_j (u - x)^k$$

for some vector of coefficients $a = (a_0, \ldots, a_D)$. As above, we will estimate the coefficients a in $g_x(u; a)$ at some target point x using weighted least squares:

$$\hat{a} = \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{ y_i - g_x(x_i; a) \right\}^2 ,$$

where $w_i \equiv w(x_i, x)$ are the kernel weights defined just above, normalized to sum to one. Derive a concise (matrix) form of the weight vector $\hat{a}$, and by extension, the local function estimate $\hat{f}(x)$ at the target value x. Life will be easier if you define the matrix X_x whose (i, j) entry is $(x_i - x)^{j-1}$, and remember that (weighted) polynomial regression is the same thing as (weighted) linear regression with a polynomial basis.

We first begin by plugging in $g_x(x_i; a)$ into the formula for $\hat{a}$, expanding terms and collecting them in matrix form.

$$\begin{aligned}
\hat{a} &= \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{ y_i - \left[a_0 + \sum_{k=1}^{D} a_j (x_i - x)^k \right] \right\}^2 \\
&= \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{ y_i - a_0 - a_1(x_i - x) - a_2(x_i - x)^2 - \ldots - a_D(x_i - x)^D \right\}^2 \\
&= \arg\min_{R^{D+1}} (y - Ra)^T W (y - Ra)
\end{aligned}$$

We are then interested in minimizing the expression. This is done by setting the derivative with respect to a of the expression to zero. The term in the brackets are expanded.

$$
\begin{aligned}
0 &= \frac{d}{da}\left[(y - Ra)^T W (y - Ra)\right] \\
&= \frac{d}{da}\left[(y^T - a^T R^T) W (y - Ra)\right] \\
&= \frac{d}{da}\left[y^T W y - y^T W R a - a^T R^T W y + a^T R^T W R a\right] \\
&= \frac{d}{da}\left[y^T W y - 2 y^T W R a + a^T R^T W R a\right]
\end{aligned}
$$

Using the hint from exercise one that $\frac{\partial (z^T A z)}{dz} = (A + A^T) z$, and $\frac{\partial b^T z}{dz} = b$.

$$
\begin{aligned}
0 &= -2 y^T W R + 2 R^T W R a \\
y^T W R &= R^T W R a
\end{aligned}
$$

We then evaluate the expression and solve for a.

$$\hat{a} = (R^T W R)^{-1} y^T W R$$

Notice that this has the same form as the linear case:

$$\hat{a} = H y$$

With the hat matrix being, $H = (R^T W R)^{-1} R^T W$. The approximate estimate at our target value is then $\hat{f}(x) = \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} \hat{a}$

(B) From this, conclude that for the special case of the local linear estimator ($D = 1$), we can write $\hat{f}(x)$ as a linear smoother of the form

$$\hat{f}(x) = \frac{\sum_{i=1}^n w_i(x) y_i}{\sum_{i=1}^n w_i(x)},$$

where the unnormalized weights are

$$
\begin{aligned}
w_i(x) &= K\left(\frac{x - x_i}{h}\right)\{s_2(x) - (x_i - x) s_1(x)\} \\
s_j(x) &= \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)(x_i - x)^j .
\end{aligned}
$$

From $\hat{f}(x) = \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} \hat{a}$, we expand the matrices for the $D = 1$ case and multiply the matrices together.

$$
\begin{aligned}
\hat{f}(x) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \left\{ \begin{bmatrix} (x_1 - x)^0 & (x_1 - x) \\ \vdots & \vdots \\ (x_n - x)^0 & (x_n - x) \end{bmatrix}^T \begin{bmatrix} w_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & w_n \end{bmatrix} \begin{bmatrix} (x_1 - x)^0 & (x_1 - x) \\ \vdots & \vdots \\ (x_n - x)^0 & (x_n - x) \end{bmatrix} \right\}^{-1} \begin{bmatrix} (x_1 - x)^0 & (x_1 - x) \\ \vdots & \vdots \\ (x_n - x)^0 & (x_n - x) \end{bmatrix}^T \begin{bmatrix} w_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & w_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \sum_{i=1}^n w_i & \sum_{i=1}^n w_i (x_i - x) \\ \sum_{i=1}^n w_i (x_i - x) & \sum_{i=1}^n w_i (x_i - x)^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^n w_i y_i \\ \sum_{i=1}^n (x_i - x) w_i y_i \end{bmatrix}
\end{aligned}
$$

We note that for a two-by-two matrix the inverse can be found as

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Evaluating the inverse and multiplying the resultant matrices together we obtain the following.

$$\begin{aligned}\hat{f}(x) &= \left[\sum_{i=1}^n w_i \sum_{i=1}^n w_i(x_i-x)^2 - \left(\sum_{i=1}^n w_i(x_i-x)\right)^2\right]^{-1} \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}\sum_{i=1}^n w_i(x_i-x)^2 & -\sum_{i=1}^n w_i(x_i-x) \\ -\sum_{i=1}^n w_i(x_i-x) & \sum_{i=1}^n w_i\end{bmatrix}\begin{bmatrix}\sum_{i=1}^n w_i y_i \\ \sum_{i=1}^n (x_i-x)w_i y_i\end{bmatrix} \\ &= \left[\sum_{i=1}^n w_i \sum_{i=1}^n w_i(x_i-x)^2 - \left(\sum_{i=1}^n w_i(x_i-x)\right)^2\right]^{-1}\left[\sum_{i=1}^n w_i(x_i-x)^2 \sum_{i=1}^n w_i y_i - \sum_{i=1}^n w_i(x_i-x)\sum_{i=1}^n w_i y_i(x_i-x)\right]\end{aligned}$$

With $s_1 = \sum_{i=1}^n w_i(x_i-x)$ and $s_2 = \sum_{i=1}^n w_i(x_i-x)^2$ we obtain the form we were seeking:

$$\hat{f}(x) = \frac{\sum_{i=1}^n w_i y_i(s_2-(x_i-x)s_1)}{\sum_{i=1}^n w_i(s_2-(x_i-x)s_1)}.$$

(C) Suppose that the residuals have constant variance σ^2 (that is, the spread of the residuals does not depend on x). Derive the mean and variance of the sampling distribution for the local polynomial estimate $\hat{f}(x)$ at some arbitrary point x. Note: the random variable $\hat{f}(x)$ is just a scalar quantity at x, not the whole function.

$$\begin{aligned}\mathrm{E}\left[\hat{f}(x)\right] &= \mathrm{E}\left[\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} Hy\right] \\ &= \begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} H\mathrm{E}[y] \\ &= \begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} Hf(x)\end{aligned}$$

$$\begin{aligned}\mathrm{var}[\hat{f}(x)] &= \mathrm{var}(\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} Hy) \\ &= \left\{\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} H\right\}^T \mathrm{var}(y)\left\{\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} H\right\} \\ &= \sigma^2\left\{\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} H\right\}^T\left\{\begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix} H\right\}\end{aligned}$$

$y_i = f(x_i) + \epsilon_i$

(D) We don't know the residual variance, but we can estimate it. A basic fact is that if x is a random vector with mean μ and covariance matrix Σ, then for any symmetric matrix Q of appropriate dimension, the quadratic form x^TQx has expectation

$$E(x^TQx) = \mathrm{tr}(Q\Sigma) + \mu^TQ\mu\,.$$

Write the vector of residuals as $r = y - \hat{y} = y - Hy$, where H is the smoothing matrix. Compute the expected value of the estimator

$$\hat{\sigma}^2 = \frac{\|r\|_2^2}{n - 2\mathrm{tr}(H) + \mathrm{tr}(H^TH)}\,,$$

and simplify things as much as possible. Roughly under what circumstances will this estimator be nearly unbiased for large n? Note: the quantity $2\mathrm{tr}(H) - \mathrm{tr}(H^TH)$ is often referred to as the "effective degrees of freedom" in such problems.

$$\begin{aligned}
\mathrm{E}\left[\hat{\sigma}^2\right] &= \mathrm{E}\left[\frac{(y-Hy)^T(y-Hy)}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)}\right] \\
&= \frac{\mathrm{E}[(y-Hy)^T(y-Hy)]}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)} \\
&= \frac{\mathrm{E}[y^Ty]-2\mathrm{E}[y^THy]+\mathrm{E}[y^TH^THy]}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}[y^Ty] &= \mathrm{tr}(\Sigma)+f(x)^Tf(x) \\
&= n\sigma^2+f(x)^Tf(x) \\
\mathrm{E}[y^THy] &= \mathrm{tr}(H\Sigma)+f(x)^THf(x) \\
&= \sigma^2\mathrm{tr}(H)+f(x)^THf(x) \\
\mathrm{E}[y^TH^THy] &= \mathrm{tr}(H^TH\Sigma)+f(x)^TH^THf(x) \\
&= \sigma^2\mathrm{tr}(H^TH)+f(x)^TH^THf(x)
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}\left[\hat{\sigma}^2\right] &= \frac{n\sigma^2+f(x)^Tf(x)-2\sigma^2\mathrm{tr}(H)-2f(x)^THf(x)+\sigma^2\mathrm{tr}(H^TH)+f(x)^TH^THf(x)}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)} \\
&= \frac{\sigma^2[n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)]+f(x)^Tf(x)-2f(x)^THf(x)+f(x)^TH^THf(x)}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)} \\
&= \sigma^2+\frac{\|f(x)-Hf(x)\|_2^2}{n-2\mathrm{tr}(H)+\mathrm{tr}(H^TH)}
\end{aligned}$$

Unbiased when the difference, $f(x)-Hf(x)$ is small

(E) Write a new R function that fits the local linear estimator using a Gaussian kernel for a specified choice of bandwidth h. Then load the data in "utilities.csv" into R. This data set shows the monthly gas bill (in dollars) for a single-family home in Minnesota, along with the average temperature in that month (in degrees F), and the number of billing days in that month. Let y be the average daily gas bill in a given month (i.e. dollars divided by billing days), and let x be the average temperature. Fit y versus x using local linear regression and some choice of kernel. Choose a bandwidth by leave-one-out cross-validation.

(F) Inspect the residuals from the model you just fit. Does the assumption of constant variance (homoscedasticity) look reasonable? If not, do you have any suggestion for fixing it?

(G) Put everything together to construct an approximate point-wise 95% confidence interval for the local linear model (using your chosen bandwidth) for the value of the function at each of the observed points x_i for the utilities data. Plot these confidence bands, along with the estimated function, on top of a scatter plot of the data.

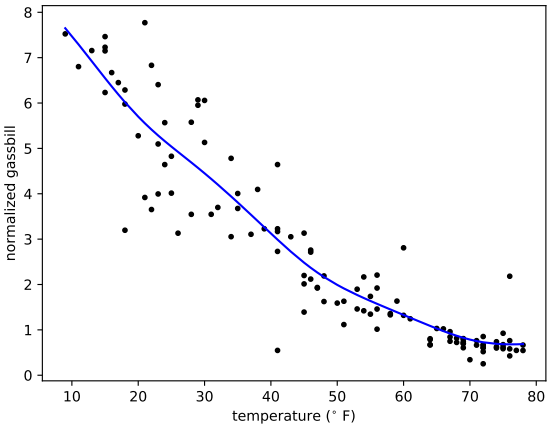

Figure 3: Fit using LOOCV, $h = 6.87$

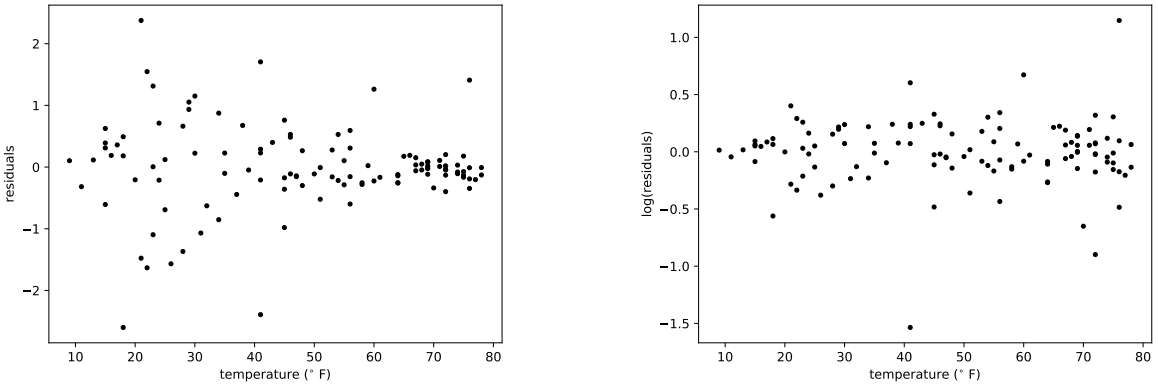

Figure 4: Residuals and log-residuals

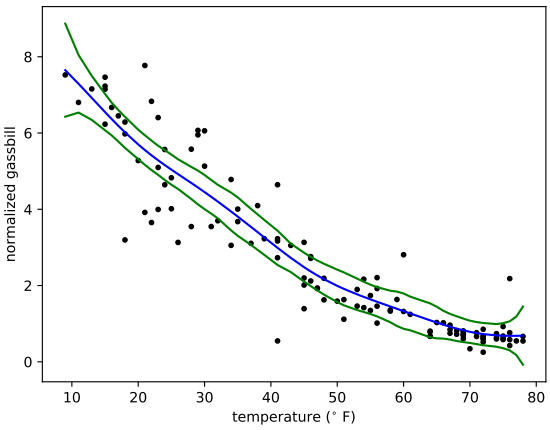

Figure 5: Fit using LOOCV, $h = 6.87$

Guassian processes

A *Gaussian process* is a collection of random variables $\{f(x) : x \in \mathcal{X}\}$ such that, for any finite collection of indices $x_1, \ldots, x_N \in \mathcal{X}$, the random vector $[f(x_1), \ldots, f(x_N)]^T$ has a multivariate normal distribution. It is a generalization of the multivariate normal distribution to infinite-dimensional spaces. The set $\mathcal{X}$ is called the index set or the state space of the process, and need not be countable.

A Gaussian process can be thought of as a random function defined over $\mathcal{X}$, often the real line or $\mathbb{R}^p$. We write $f \sim \mathrm{GP}(m, C)$ for some mean function $m : \mathcal{X} \to \mathbb{R}$ and a covariance function $C : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$. These functions define the moments of all finite-dimensional marginals of the process, in the sense that

$$E\{f(x_1)\} = m(x_1) \quad \text{and} \quad \mathrm{cov}\{f(x_1), f(x_2)\} = C(x_1, x_2)$$

for all $x_1, x_2 \in \mathcal{X}$. More generally, the random vector $[f(x_1), \ldots, f(x_N)]^T$ has covariance matrix whose (i, j) element is $C(x_i, x_j)$. Typical covariance functions are those that decay as a function of increasing distance between points x_1 and x_2. The notion is that $f(x_1)$ and $f(x_2)$ will have high covariance when x_1 and x_2 are close to each other.

(A) Read up on the Matern class of covariance functions. The Matern class has the *squared exponential* covariance function as a special case:

$$C_{SE}(x_1, x_2) = \tau_1^2 \exp\left\{-\frac{1}{2}\left(\frac{d(x_1, x_2)}{b}\right)^2\right\} + \tau_2^2 \delta(x_1, x_2)\,,$$

where $d(x_1, x_2) = \|x_1 - x_2\|_2$ is Euclidean distance (or just $|x - y|$ for scalars). The constants (b, τ_1^2, τ_2^2) are often called *hyperparameters*, and $\delta(a, b)$ is the Kronecker delta function that takes the value 1 if $a = b$, and 0 otherwise. But usually this covariance function generates functions that are "too smooth," and so we use other covariance functions in the Matern class as a default.

Let's start with the simple case where $\mathcal{X} = [0, 1]$, the unit interval. Write a function that simulates a mean-zero Gaussian process on $[0, 1]$ under the Matern(5/2) covariance function. The function will accept as arguments: (1) finite set of points $x_1, \ldots, x_N$ on the unit interval; and (2) a triplet (b, τ_1^2, τ_2^2). It will return the value of the random process at each point: $f(x_1), \ldots, f(x_N)$.

Use your function to simulate (and plot) Gaussian processes across a range of values for b, τ_1^2, and τ_2^2. Try starting with a very small value of τ_2^2 (say, 10^{-6}) and playing around with the other two first. On the basis of your experiments, describe the role of these three hyperparameters in controlling the overall behavior of the random functions that result. What happens when you try $\tau_2^2 = 0$? Why? If you can fix this, do—remember our earlier discussion on different ways to simulate the MVN.

Now simulating a few functions with a different covariance function, the Matérn with parameter 5/2:

$$C_{M52}(x_1, x_2) = \tau_1^2 \left\{1 + \frac{\sqrt{5}d}{b} + \frac{5d^2}{3b^2}\right\} \exp\left(\frac{-\sqrt{5}d}{b}\right) + \tau_2^2 \delta(x_1, x_2)\,,$$

where $d = \|x_1 - x_2\|_2$ is the distance between the two points x_1 and x_2. Comment on the differences between the functions generated from the two covariance kernels.

- b: Controls the smoothness of the resulting function. As b increases, the smoothness increases until it approaches a constant.
- τ_1^2: appears to control how much the resulting function varies from the mean at a given point.
- τ_2^2: seems to increase the noise as its value increases.

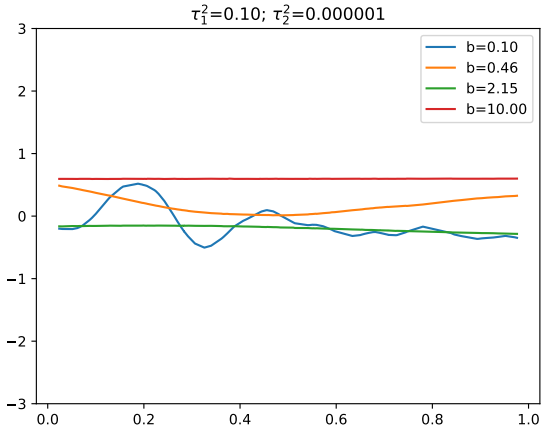

Figure 6: Varied b for Matern(5,2) covariance function

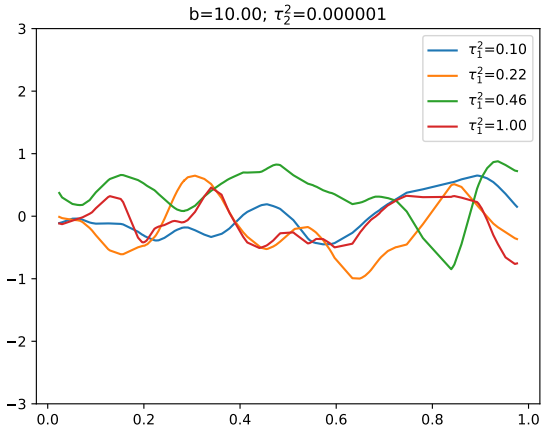

Figure 7: Varied τ_1^2 for Matern(5,2) covariance function

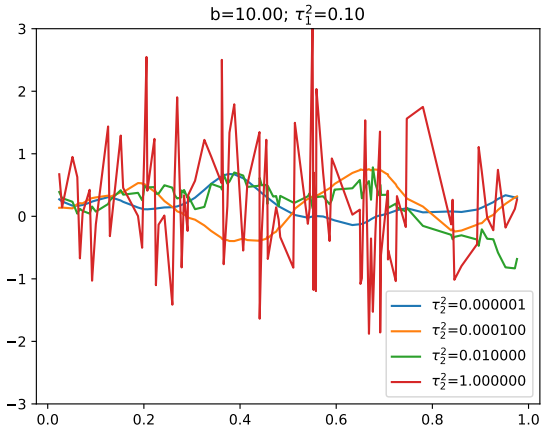

Figure 8: Varied τ_2^2 for Matern(5,2) covariance function

(B) Suppose you observe the value of a Gaussian process $f \sim \mathrm{GP}(m, C)$ at points $x_1, \ldots, x_N$. What is the conditional distribution of the value of the process at some new point $x^\star$? For the sake of notational ease simply write the value of the (i, j) element of the covariance matrix as $C_{i,j}$, rather than expanding it in terms of a specific covariance function.

$$[f(x_1), ..., f(x_N)]^T \sim \mathrm{N}(m, C)$$

$$\mathrm{N}(m, C) = \mathrm{N}\left(\begin{bmatrix} m_1 \\ \vdots \\ m_n \end{bmatrix}, \begin{bmatrix} C_{11} & \ldots & C_{1n} \\ \vdots & \ddots & \vdots \\ C_{n1} & \ldots & C_{nn} \end{bmatrix}\right)$$

$$[f(x_1), ..., f(x_N), f(x^*)]^T \sim \mathrm{N}\left(\begin{bmatrix} m_1 \\ \vdots \\ m_n \\ m^* \end{bmatrix}, \begin{bmatrix} C_{11} & \ldots & C_{1n} & C_{1*} \\ \vdots & \ddots & \vdots & \vdots \\ C_{n1} & \ldots & C_{nn} & C_{n*} \\ C_{*1} & \ldots & C_{*n} & C_{**} \end{bmatrix}\right)$$

$$\sim \mathrm{N}\left(\begin{bmatrix} \mathbf{m} \\ m^* \end{bmatrix}, \begin{bmatrix} C(\mathbf{x}, \mathbf{x}) & C(\mathbf{x}^*\mathbf{x}) \\ C(\mathbf{x}^*\mathbf{x})^T & C(\mathbf{x}^*, \mathbf{x}^*) \end{bmatrix}\right)$$

$$x_1 | x_2 \sim \mathrm{N}(\mu_1 + \Sigma_{22}^{-1}\Sigma_{12}^T(x_2 - \mu_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^T)$$

$$f(x^*)|f(x_1), ..., f(x_N) \sim \mathrm{N}\left(m^* + C(\mathbf{x}, \mathbf{x})^{-1}C(\mathbf{x}^*\mathbf{x})^T(f(x_1), ..., f(x_N) - \mathbf{m})\right), C(\mathbf{x}^*, \mathbf{x}^*) - C(\mathbf{x}^*\mathbf{x})C(\mathbf{x}, \mathbf{x})^{-1}C(\mathbf{x}^*\mathbf{x})^T$$

(C) Prove the following lemma.

Lemma 1 *Suppose that the joint distribution of two vectors y and θ has the following properties: (1) the conditional distribution for y given θ is multivariate normal, $(y \mid \theta) \sim N(R\theta, \Sigma)$; and (2) the marginal distribution of θ is multivariate normal, $\theta \sim N(m, V)$. Assume that R, Σ, m, and V are all constants. Then the joint distribution of y and θ is multivariate normal.*

The model for y can be written as

$$y = R\theta + \epsilon$$

where

$$y \sim \mathrm{N}(R\theta, \Sigma)$$
$$\epsilon \sim \mathrm{N}(m, V)$$

We can then show that the system $y = R\theta + \epsilon$ and $\theta = \theta + 0$ can be written

$$\begin{bmatrix} y \\ \theta \end{bmatrix} = \begin{bmatrix} R \\ I \end{bmatrix} \theta + \begin{bmatrix} I \\ 0 \end{bmatrix} \epsilon$$

This is an affine transformation of multivariate normals, hence it is also multivariate normal.

A Kernel Smoothing Code

```
from __future__ import division
import numpy as np
from scipy.optimize import minimize
from numpy.linalg import inv

class kernels:
    """
    This class contains kernel functions for kernel smoothing
    """
    def __init__(self):
        pass

    def uniform(x):
        """
        Parameters
        ----------
            x: float
                argument to be evaluated

        Returns
        ----------
            k: float
                uniform kernel evaluation at x
                .. math:: k(x) = 1/2 I(x), with I(x) = 1, if |x| \\leq 1, 0 otherwise
        """
        if np.abs(x) <= 1:
            k = 0.5
        else:
            k = 0
        return k

    def gaussian(x):
        """
        Parameters
        ----------
            x: float
                argument to be evaluated

        Returns
        ----------
            k: float
                gaussian kernel evaluation at x
                .. math:: k(x) = \\frac{1}{\\sqrt{2 \\pi}} e^{-x^2/2}
        """
        return 1/np.sqrt(2*np.pi)*np.exp(-x**2/2)

class kernel_smoother:
    """
    This class returns a vector of smoothed values given feature and response vectors
    """
    def __init__(self, x, y, x_star, kernel='gaussian', h=1, D=1):
        """
        Parameters
        ----------
            x: float
                Feature vector

            y: float
                Response vector

            x_star: float
                Scalar or vector to be evaluated

            kernel: str (optional)
                Kernel type to be used: Available kernels are gaussian,
```

```
            h: float (optional)
                Bandwidth

            D: int (optional)
                Order of polynomial smoother
        """
        self.x = x
        self.y = y
        self.x_star = x_star
        self.h = h
        self.D = D
        self.kernel=kernel

        if x_star.shape != x:
            raise Warning('Feature vector and evaluation vector are not the same length. Residuals ↩
                will not be calculated.')

    def local_general(self):
        """
        Returns
        ----------
            y_star: float, len(x_star)
                Predictor for x_star,
                .. math:: y_i^* = [1 0 ...] H_i y
                .. math:: w(x_i, x^*) = \\frac{1}{2}K( \\frac{x_i - x^*}{h}), \\sum_{i=1}^n w(x_i, ↩
                    x^*) = 1

        Attributes
        ----------
            sigma: float, len(x_star)
                Standard error at x_star

            residuals: float, len(x_star)
                residuals of prediction from data (only calculated if len(y_star) == (len(y)))
                .. math:: r_i = y_i - \\hat{y}_i

            Hat_matrix: list
                first elements in hat matrix for a given x_star
        """
        # Instantiate vectors and matrices for storing calculations
        self.y_star = np.zeros(self.x_star.shape)
        self.sigma = np.zeros(self.x_star.shape)
        self.residuals = np.zeros(self.x_star.shape)
        self.Hat_matrix = []

        # Vector for determining y_star
        e_1 = np.append(1, np.zeros(self.D))

        # Iterate through each value in y_star
        for i in range(self.y_star.shape[0]):

            # Instantiate a weights vector
            weights = np.zeros(self.x.shape)

            # Instantiates X matrix
            X = np.transpose(np.ones((self.x.shape[0], self.D+1)))

            # Populates X matrix given the order of the smoother, D
            for d in range(self.D):
                X[d+1] = (self.x - self.x_star[i])**(d+1)
            X = np.transpose(X)

            # Iterates through feature/response vectors
            for j in range(self.x.shape[0]):
                weights[j] = 1/self.h * getattr(kernels, self.kernel)((self.x[j] - self.x_star[i])/↩
                    self.h)
```

```
            # Normalizes weights such that \sum_{i=1}^n w(x_i, x^*) = 1
            weights = weights/weights.sum()

            # Construct weights matrix
            W = np.diag(weights)

            # Calculate hat matrix; H = (X^T W X)^{-1} X^T W
            H = (inv(np.transpose(X) @ W @ X) @ np.transpose(X) @ W)

            # Append hat matrix with current H matrix (needed for LOOCV)
            self.Hat_matrix.append(e_1 @ H)

            # \hat{f}(x^*) = e_1 H y
            self.y_star[i] = e_1 @ H @ self.y

            # Calculates residuals and confidence interval if y_star and y shapes are equivalent
            if self.y_star.shape == self.y.shape:
                self.residuals[i] = self.y[i] - self.y_star[i]

        # Residual sum of squared errors
        rss = (self.residuals**2).sum()
        # Approximate standard error of fit
        sigma_hat = rss/(len(self.y_star)-1)

        # Calculates approximate standard error
        for i in range(self.sigma.shape[0]):
            self.sigma[i] = np.sqrt(np.transpose(self.Hat_matrix[i]) @ (self.Hat_matrix[i])*↩
                sigma_hat)

        return self.y_star

    def MSE(self, y_test):
        """
        Parameters
        ----------
            y_test: float
                Response vector

        Returns
        ----------
            MSE: float
                Average squared error for y_test given y
                .. math:: MSE \\approx \\frac{1}{n}\\sum(y_test - y_star)**2
        """
        self.y_test = y_test
        return np.mean((y_test - self.y_star)**2)

    def MSE_optimization(self, y_test):
        """
        Parameters
        ----------
            y_test: float
                Response vector

        Returns
        ----------
            h_star: float
                Uses BFGS minimization method to minimize the MSE by varying h_star
        """
        self.y_test = y_test
        def func(X):
            Y = kernel_smoother(self.x, self.y, self.x_star, kernel=self.kernel, h=X, D=self.D)
            Y.local_general()
            return Y.MSE(self.y_test)
        h_star = minimize(func, 1)
        self.h = h_star.x
        return h_star.x
```

```
    def LOOCV(self):
        """
        Returns
        ----------
            loocv: float
                Leave one out cross validation statistic (LOOCV)
                .. math: loocv = \\sum_{i=1}^n (\\frac{y_i - \\hat{y}_i}{1-H_{ii}})^2
        """
        loocv = np.zeros(len(self.y))
        for i in range(len(loocv)):
            loocv[i] = ((self.y[i] - self.y_star[i])/(1 - self.Hat_matrix[i][i]))**2
        return loocv.sum()

    def LOOCV_optimization(self):
        """
        Returns
        ----------
            h_star: float
                Uses BFGS minimization method to minimize the LOOCV by varying h_star
        """
        def func(X):
            Y = kernel_smoother(self.x, self.y, self.x_star, kernel=self.kernel, h=X, D=self.D)
            Y.local_general()
            return Y.LOOCV()
        h_star = minimize(func, 0.7)
        self.h = h_star.x
        return self.h
```